POLITECNICO DI MILANO

SOFTWARE ENGINEERING II

eMall- e-Mobility for All

Design Document

Version 1.0

[redacted]

Contents

1. Introduction

1.1 Purpose

The main purpose of the present document is to support the development team in the realization of the system to be. It provides an overall description of the adopted system architecture and a breakdown of the various components down to a significant extent, also describing their interactions with each other. On top of that, the Design Document (DD) describes the implementation, integration and testing plans which are defined keeping into account priority, required effort, and impact of the single components on the stakeholders

1.2 Scope

eMall- e-Mobility for All (abbreviated eMall) is a software system that allows customers to reserve and also to manage a booking in a recharge station, in order to reduce the impact on the user's daily life in recharging the EV.
The objective is to allow users to recharge the EV without too much difficulty in managing the whole process. Users reserve a recharge by *booking a recharge* in a specific socket and *timeslot*.
The system manages the recharge by allowing to book a recharge only in the available time slots of each station, i.e the time slots for which the station capacity has not been reached, so that users cannot experience a downtime since the station has no available sockets.
Moreover, during the “booking a recharge” process, users are allowed to select specific time slots and of a station where they want to recharge. That information is exploited by the system in order to show to the users only the available time slots and socket for each station.
Those first few lines just give an overview of the system and are necessary to highlight the fact that it can be a very useful tool for allowing customers to fulfill their need for the recharging process. Further details on the above can be found in the Requirements Analysis and Specification Document (RASD).

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Booking**: The action of booking a recharge station for a timeframe performed by a user.
- **TimeFrame**: A time interval during which the recharge station is booked to a specific User.
- **Smart User/User**: User who has got a smartphone with the e-mall app. She/He's able to manage bookings by herself/himself.
- **QR-code**: A machine-readable code consisting of a sequence of black and white squares.
- **QR-reader**: electronic device which scans QR-codes. It also has a small screen.
- **Internal status of a charging station**: it includes the amount of energy available in its batteries, if any, the number of vehicles being charged and, for each charging vehicle, the amount of power absorbed and the time left to the end of the charge
- **External status of a charging station**: it includes the number of charging sockets available, their type such as slow/fast/rapid, their cost, and, if all sockets of a certain type are occupied, the estimated amount of time until the first socket of that type is freed
- **Application user interface**: It defines the operations that can be invoked on the component which offers it.
- **Demilitarized zone:** A middle ground between an organization's trusted internal network and an untrusted, external one

1.3.2 Acronyms

- **e-Mall** : e-Mobility for All.
- **RASD** : Requirement Analysis and Specification Document.
- **CPMS** : Charge Point Management System.
- **CPOs** : Charging Point Operators.
- **DSOs** : Distribution System Operators.
- **UI** : User interface.
- **DMZ**: DeMilitarized Zone

1.3.3 Abbreviations

- [Gn] - the n-th goal of the system
- [WPn] - the n-th world phenomena
- [SPn] - the n-th shared phenomena
- [UCn] - the n-th use case
- [Rn] - the n-th functional requirement
- [EV] - electric vehicle
- DB - Database
- API - Application Programming Interface
- DMZ - Demilitarized Zone

1.4 Revision history

- Version 1.0 (January 08th 2023 : first version);

1.5 Reference Documents

This document is strictly based on:

- The specification of the RASD and DD assignment of the Software Engineering II course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2022/2023;
- Slides of Software Engineering 2 course on WeBeep;
- Official link of Enel: https://www.enelx.com/uk/en.html, to get information on recharge time
- DD Sample from A.Y. 2020-2021

1.6 Document Structure

Mainly the current document is divided in 4 chapters, which are:

1. **Introduction**: The first chapter includes the introduction in which is explained the purpose of the document, then, a brief recall of the concepts introduced in the RASD is given. Finally, important information for the reader is given, i.e. definitions, acronyms, synonyms and the set of documents referenced.
2. **Architectural Design**: it includes a detailed description of the architecture of the system, including the high level view of the elements, the software components of CLup, a description through runtime diagrams of various functionalities of the system and, finally, an in-depth explanation of the architectural pattern used.
3. **User Interface Design**: are provided the mockups of the application user interfaces, with the links between them to help in understanding the flow between them.
4. **Requirements Traceability**: it describes the connections between the requirements defined in the RASD and the components described in the first chapter. This is used as proof that the design decisions have been taken with respect to the requirements, and therefore that the designed system can fulfill the goals.
5. **Implementation,Integration and Test Plan**: it describes the process of implementation, integration and testing to which developers have to stick in order to produce the correct system in a correct way.
6. **Effort spent**
7. **References**: it contains the references to any documents and to the Software used in this document.

2. Architectural design

2.1 Overview

In this section is given an overview of the architectural elements that compose the system, their interaction and a description of the replication mechanism chosen for the system in order to make it distributed.

2.1.1 High level view

The eMall system-to-be is a distributed system with a three-tier architecture (Figure 2.1). It includes the Presentation layer for formatting the information to be shown, the Application layer which encapsulates the business logic of the application and the Data layer that manages the access to the stored data. This stratification is not only logical but also physical. In fact, each layer corresponds to a machine or a cluster of machines that host the meaningful components of both hardware and software for covering the assigned role. Each layer can communicate only with the adjacent one, this means that the data layer and the presentation layer are never going to communicate directly

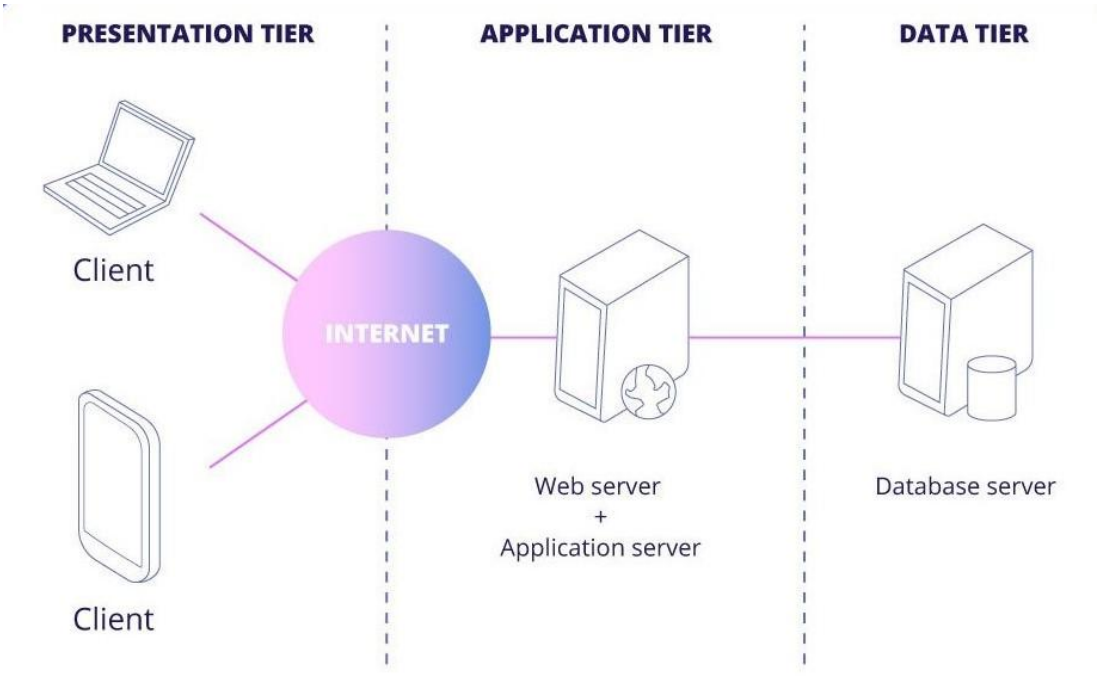


Figure 2.1 - High level system architecture

The communication between client and server involves different components depending on the consumer device's type .More specifically, the system can be accessed via browser by the CPO and via mobile application by a user (or also a CPO).

On the server side we distinguish two main components that are application server and web server. The latter handles the HTTP requests sent by browsers and returns web pages in response. Besides, the web server eventually communicates with the CPMS application server if, in order to satisfy such requests, computations or data from the database are needed; in fact, the web server and dbs are not directly connected.
Conversely, requests from the mobile application, which already encapsulates the user interface, are addressed to the eMSP application server that returns the requested information without the need of formatting them.
Users using the smartphone application, are notified when the recharge they booked is approaching.Provided such information, it is possible to notify the user when it is time to go, i.e. the recharge is approaching.
Furthermore, the CPMS and eMSP application servers communicate with each other thanks to OCPI (Open Charge Point Interface) that is an open protocol used for connections between charge station operators and service providers. This protocol facilitates automated roaming for EV drivers across several EV charging networks. This interface underpins the affordability and accessibility of charging infrastructure for the EV owners, allowing drivers to charge on several networks. The protocol provides accurate charge station data such as location, accessibility, and pricing, and it takes into account real-time billing and mobile access to charge stations.
Finally, in order to provide the requested set of functionalities the CPMS application server and eMSP application server need to be connected to a database from which retrieving all the information about users, CPOs, recharges, bookings and time slots availability. But the eMSP Application Server also needs to communicate with the Google Maps service, so to provide the search station functionality and with the Payment Authority to process the charging payment. The CPMS Application Server instead needs to communicate with the DSO to process the DSO changing and also with the socket software for both the charge of the vehicle and managing the station's battery.
Further details on the adopted architecture and used patterns are described in chapter 2.6
In order to guarantee a sufficient level of security, a front-end firewall is installed on the router which connects the internal network of eMall up with the internet. An additional back-end firewall could be eventually installed in between the application tier and the data layer, so as to offer a higher degree of security.

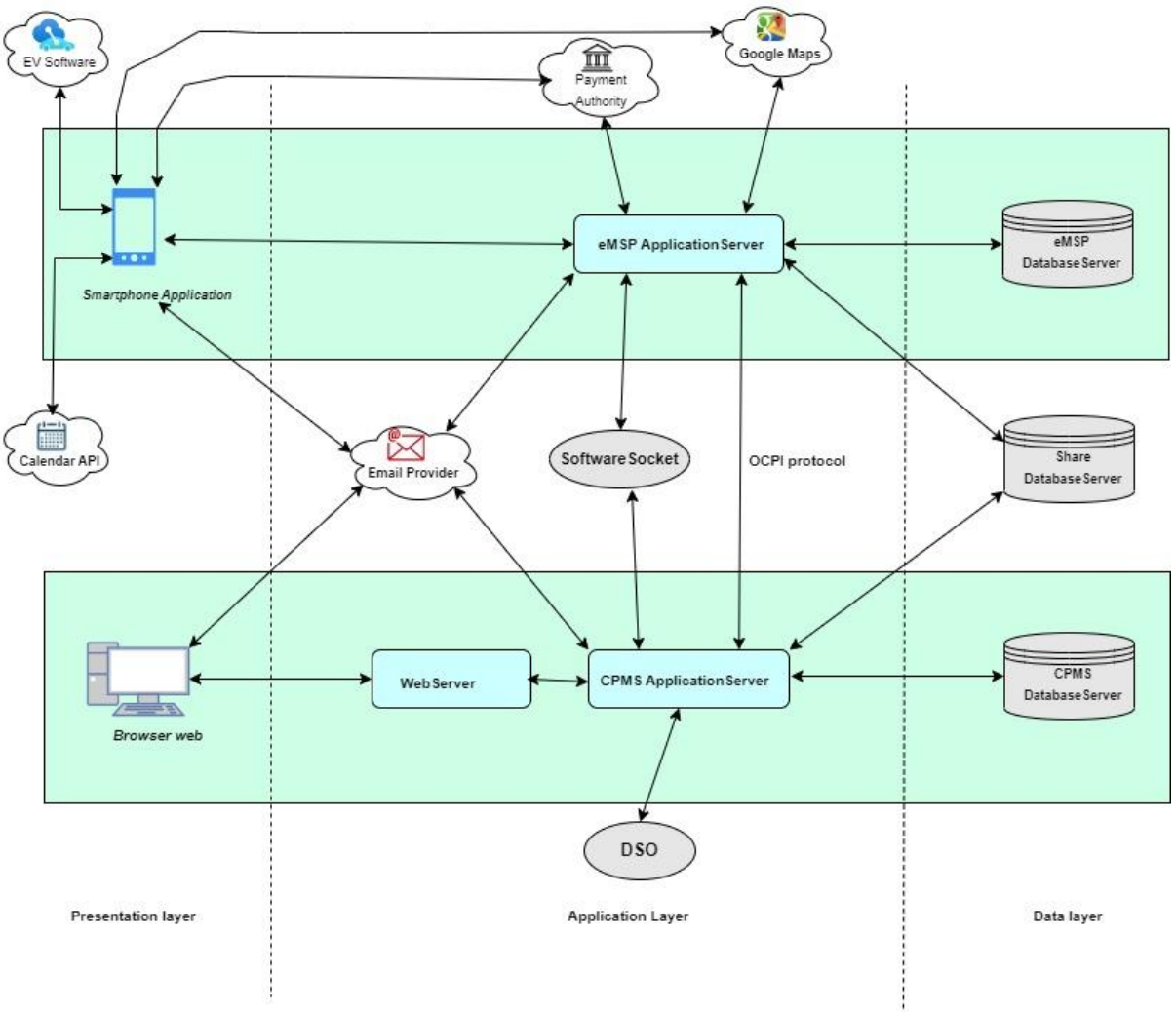


Figure 2.2 - High level system architecture 2

2.1.2 Distributed view

Most of the elements represented in Figure 2.2 are going to be replicated in order to fulfill the performance and availability requirements. Figure 2.3 and 2.4 highlight the distributed elements of the architecture.

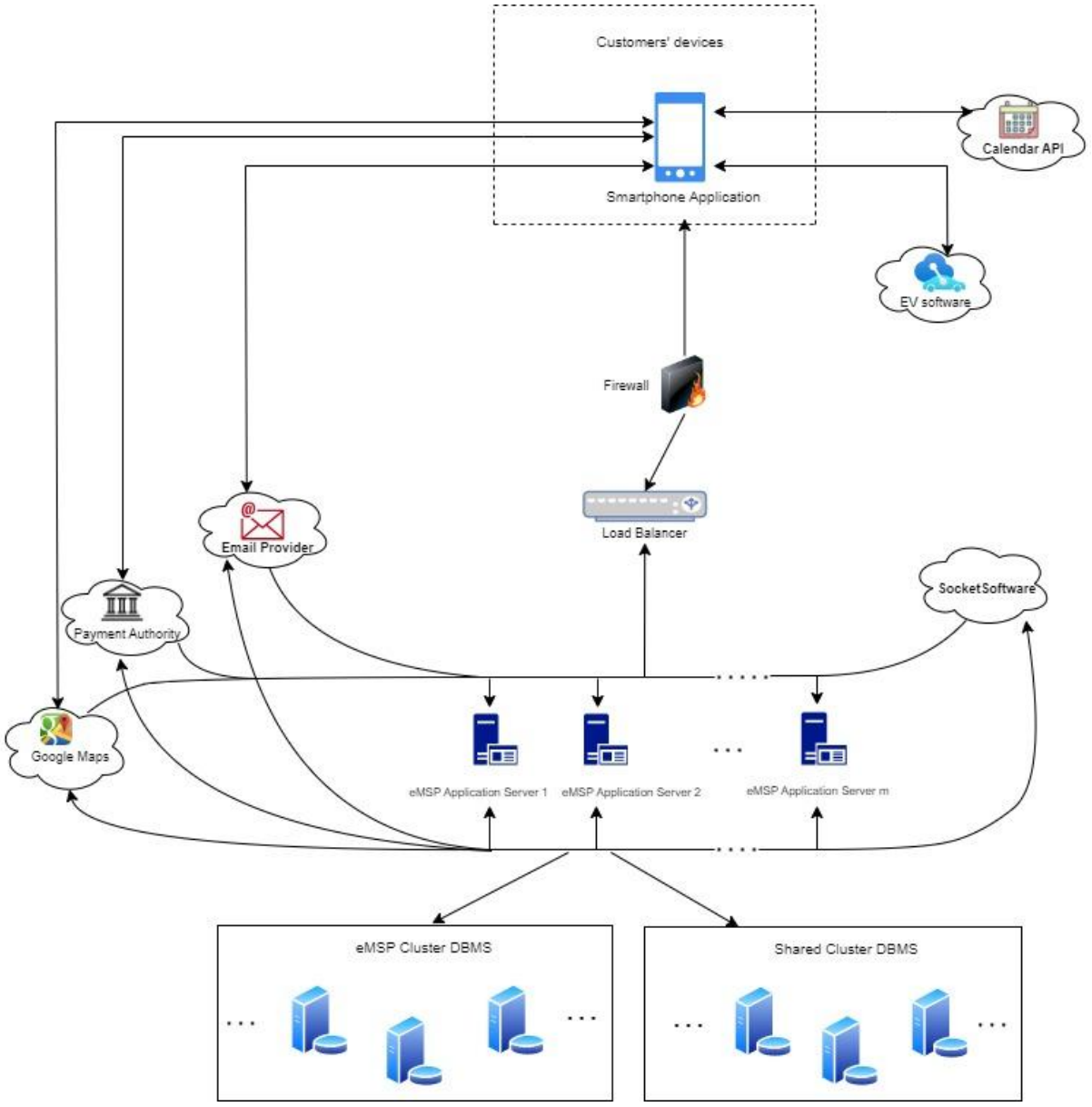


Figure 2.3 - Distributed eMSP architecture

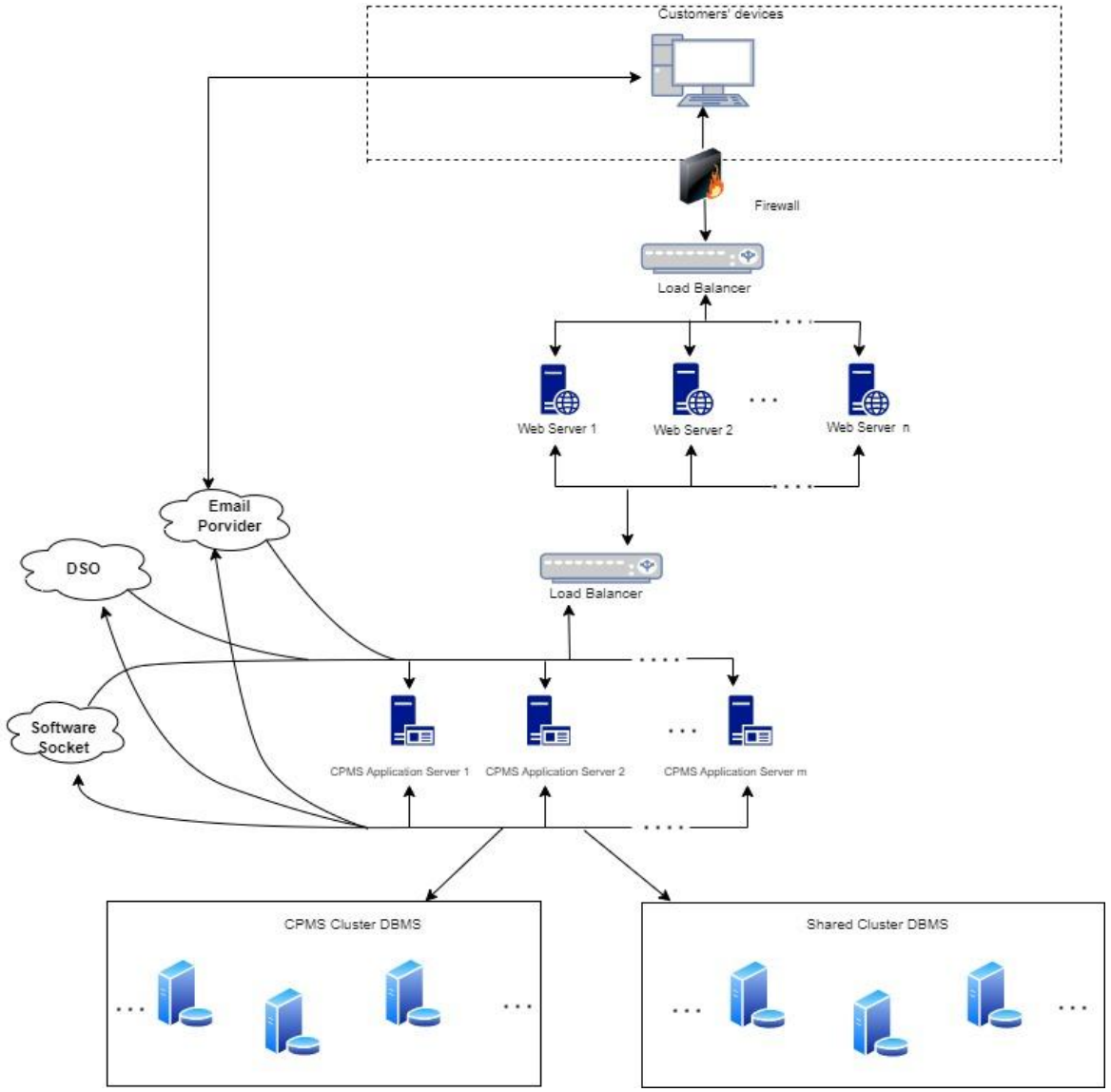


Figure 2.4 - Distributed CPMS architecture

Load balancing

The distributed architectures exploit two load balancers to dispatch the requests to web servers and application servers on the CPO side. On the user side, a load balancer is used to dispatch the request to the eMSP application server.

On the CPMS side, all the requests coming from web browsers are captured by the first load balancer which redirects the request to the more suitable web server available to handle it, i.e. the server which minimizes the response time. Then the web server manipulates the request and forwards it to the second load balancer. The second load balancer dispatches requests from web servers to the CPMS application servers according to the same load balancing criteria adopted by the first one. On the other side(eMSP), all the requests coming from smartphones are captured by the load balancer which redirects the request to the more suitable eMSP application server available to handle it, i.e. the server which minimizes the response time.

Application servers

The application tier, also known as the logic tier or middle tier, is where information collected in the presentation tier is processed (sometimes against other information in the data tier) using business logic, which means a specific set of business rules. The application tier can also add, delete or modify data in the data tier, in fact, this tier is able to communicate with the data tier by using API calls.

Clustered DBMS

Regarding the DBMS, it should be clustered. Any service that provides solid clustered DBMS can be adopted for eMall, as long as the consistency is guaranteed and the performances are sufficient to fulfill the queries in a reasonable time. Finally, all the packets incoming inside the eMall network, i.e. the set of application/web servers and DBMS, are filtered by a pair of firewalls before being sent to the load balancers.

2.2 Component view

In Figure 2.5 and 2.6 are represented the component diagrams of the system. All the components in yellow are elements of the application server . WebServer is colored in blue and it represents the element not belonging to the application server, yet still being part of the application layer. The elements colored in red, instead, represent the components directly provided to users for interacting with the system, i.e. the ones that belong to the Presentation tier. The only element colored in green is the DBMS, which is the only one belonging to the data tier. Finally, GoogleMaps, Payment Authority, SocketSoftware, DSO, EmailProvider, EVSoftware and CalendarAPI components are represented in purple to highlight the fact that they are elements provided by third party, hence not belonging strictly to eMall.

For the sake of simplicity we have split the whole component diagram of the system in two diagrams which represent respectively the components that are part of the eMsp and those that are part of the CPMS.

The NotificationManager component is coloured in brown to highlight the fact that it is outside the eMsp and CPMS but it is used by both.

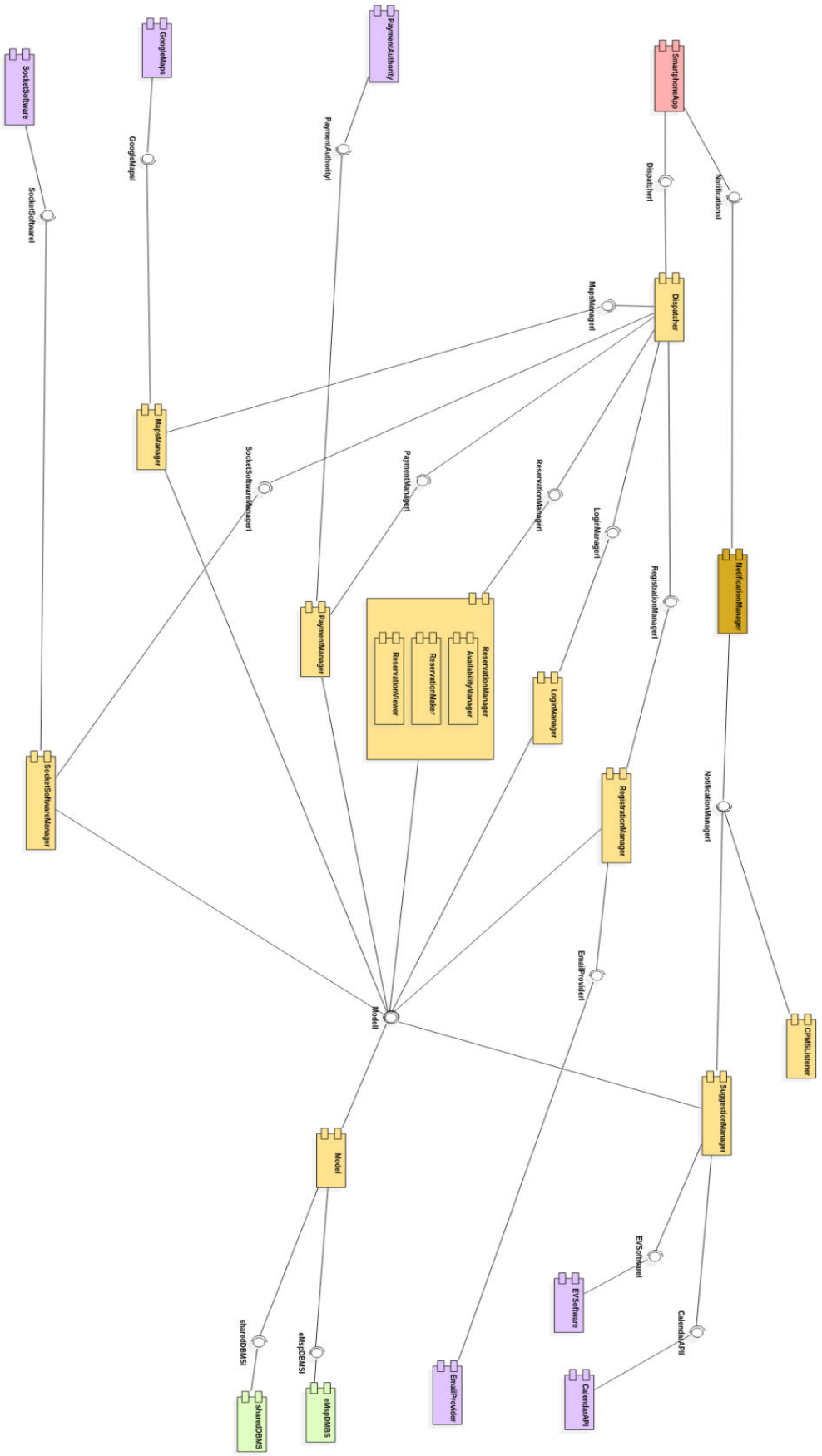


Figure 2.5 - eMsp component diagram

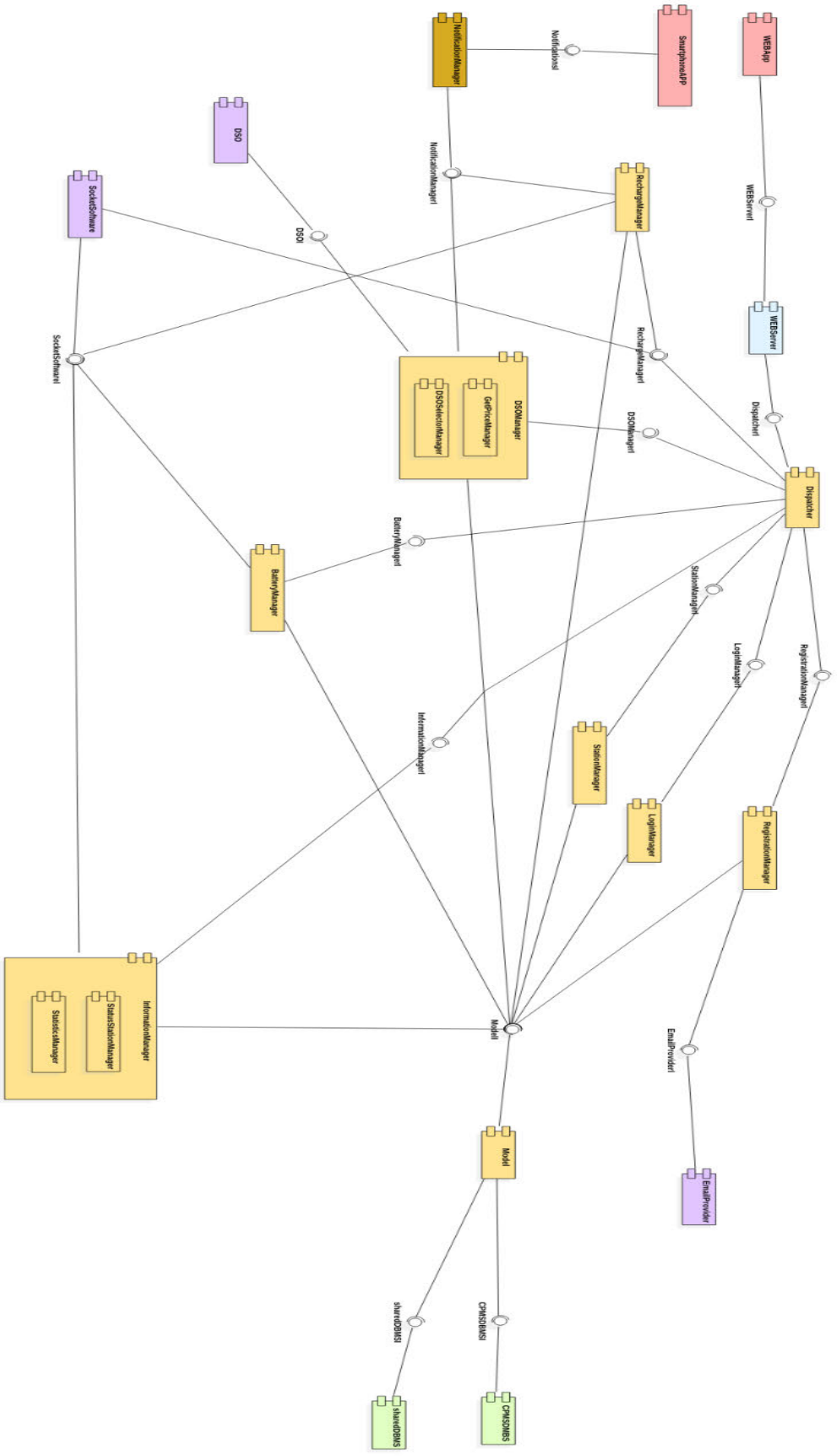


Figure 2.5 - CPMS component diagram

- The **WebServer** component is designed to satisfy all the requests from the eMall web application accessed by the CPO devices' browser. It sends on the requests to the dispatcher (contained in the application server) which, after forwarding them to the competent components, returns the answer web pages that are finally forwarded to the CPO by the web server itself.
- The **SharedDBMS** component allows the creation and manipulation of the database that is necessary for the functioning of eMall. The db contains all the relevant shared data to the application and it resolves the queries received from the model component with the requested data.
- The **CPMSDBMS** component contains all the relevant data that are exclusive for the CPO like personal info and the stations managed.
- The **eMspDBMS** component contains all the relevant data that are exclusive for the User like personal info and the vehicles owned.
- The **Dispatcher** receives all the requests from browsers together with all the calls to methods made by the mobile applications. Next, it forwards them to the competent component and, once the responses are ready, the dispatcher itself returns the answers to the clients.
- The **Model** component is used to store the data that are part of the persistent state of the system (which in our case is stored on a database) when this is loaded into the application. Moreover, it provides all the methods that are necessary to access those data and it also implements the logic to manipulate them. In other words: whenever some data contained in the database is needed by any component of the system, for any kind of computation, they delegate the task of retrieving such information to the model component. It not only retrieves them but it also stores them in ad-hoc data structures so that they are ready to be used for the required computations. The model component is the only one interacting with the dbms service one.
- The **LoginManager** component allows both CPOs and users to enter their reserved page if the credentials they provide are correct. In general it has the role of granting the privacy of the customers' data by denying the access to unauthorized customers.
- The **RegistrationManager** component which contains the service used to manage the registration of CPOs and users to eMall. Based on the type of customer, this component will provide a specific form to be filled in to be able to register with the appropriate information such as for example a CPO needs to enter the managed workstations while these data are not necessary for a user
- The **MapsManager** component handles the dialog with the user when he is searching for a station on the map. More precisely: every time the user inserts an address or an ID , the MapsManager communicates with GoogleMaps via the dedicated APIs and retrieves the updated map for the user.
- The **PaymentManager** component handles the dialog with the user when he has to handle the payment for a recharge . More precisely: every time the user inserts a new payment method , the PaymentManager communicates with payment authority

via the dedicated APIs and retrieves the correctness of the payment method of a user.
Moreover if the user's CC is valid the PaymentManager communicates through the OCPI protocol to the RechargeManager to start the charging, the same is if the QRcode is valid so that the RechargeManager communicates to the payment manager to start checking validity of the user's credit card and also if charging is ended the RechargeManager tells the PaymentAuthority to process the payment.

- The **SocketSoftwareManager** component is responsible for mediating the current number of vehicles attached to the station. Every time a new ticket is scanned, it determines whether the user is allowed to attach the vehicle or not, depending on the validity of the provided ticket.
- The **CPMSListener** is responsible for waiting for notifications through the OPCI protocol from the RechargeManager about the ending of the charge to send to the SmartphoneApp.
- The **RechargeManager** component is responsible, once a user is authorized to connect to the socket the EV, to start charging a vehicle by communicating with the Socket software in order to start the recharge. It also memorizes the starting time of the recharge likewise it saves the ending time when the user disconnects the EV from the station or the timeslot ends. Both starting time and ending time as well as the cost of a recharge, the userID, the stationID and the type of the socket used are saved into the database by the RechargeManager through the model component. Furthermore, this component uses NotificationManager to send a notification about the last recharge that has to be sent to the user's smartphone. This component is responsible also for setting the socket software as busy in the SharedDBMS.
  This component uses the OCPI protocol to send a notification to the CPMSListener.
- The **SuggestionManager** component is responsible for the smarter part of the eMSP. Once a user allows this option, this component gathers continuously information from both smartphone and EV(thanks to the bluetooth connection between smartphone and EV software). Then, proactively suggests to the user when to go and charge the vehicle, depending on the status of the battery, the schedule of the user, the special offers made available and the availability of charging slots at the identified stations. This component uses NotificationManager to manage the notification that has to be sent to the user's smartphone.
- The **WebApp** is the web interface used by the CPO to access the system. Basically it communicates with the system via HTTP requests to the WebServer, for this reason it can simply be executed on a web browser.
- The **SmartphoneApp** is the application for mobile devices provided to users for accessing the system via smartphone/tablet. It is required to be installed on compatible devices and its communication with the system involves directly the application server, without passing through the WebServer, hence interacting directly with the Dispatcher.

- The **ReservationManager** is the component designated, as the name suggests, to manage all the tasks related to the booking process. In particular it checks the availability of both socket and time slots required by the user via the sub-component *AvailabilityManager*. The time slots instances are created and stored when the first booking that requires such a time slot is reserved, which means that their creation is managed by the ReservationManager via the *ReservationMaker* sub-component. The latter is the core of the ReservationManager and is the element that really creates the reservations inside the system, both by calling the model to write the reservation details and by generating the QR-codes necessary for the customers. Then there is another sub-component, the *ReservationViewer* that is responsible for doing queries to SharedDBMS to show to the user all his bookings and reservations (i.e. both the booking already occurred and not).
- The **NotificationManager** is the component that is in charge of handling the notifications generated by the system. Every time the system has to send a notification to the user's smartphone this component manages those notifications.
- The **InformationManager** is the component designated, as the name suggests, to retrieve all the information inherent to the stations requested by CPO while they use the system. In particular, it queries the SocketSoftware system and it gathers the internal and external status in each station managed by the operator via the sub-component *StatusStationManager*.
  It also retrieves the information related to the number of EV attached to a station in a specific period, the details of reservation associated with the managed stations and the information about the recharged vehicle via the sub-component *StatisticsManager* and also provides answers to the queries it receives, by communicating with the Model.
- The **DSOManager** is the component that manages all the tasks related to the DSO information requested by CPO while they use the system. In particular, it gathers the information regarding the prices offered by different DSOs via the sub-component *GetPriceManager.* Once the operator chooses, the *DSOSelectorManage*r sub-component stores this information. Moreover, the *DSOSelectorManager* sub-component uses NotificationManager to send the notification that has to be sent to the user's smartphone such as the changing of price due to DSO changing if the user has a reservation.
- The **BatteryManager** is the component that allows the CPO to decide from where to get energy for charging, in particular it communicates with the SocketSoftware to set how to acquire energy.
- The **StationManager** is the component that allows the operator to manage the data of a station. In particular, it allows an operator to add a new station to the system, either to modify or to add information about a station and also to set special offers for stations.

2.3 Deployment view

In this chapter is described the deployment view for eMall(Figure 2.7 and Figure 2.8). This view describes the execution environment of the system, together with the geographical distribution of the hardware components that executes the software composing the application.

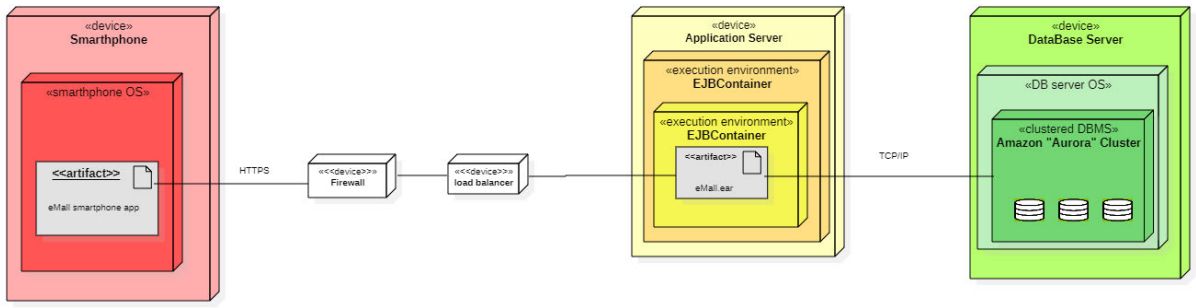


Figure 2.7 - eMSP Deployment view

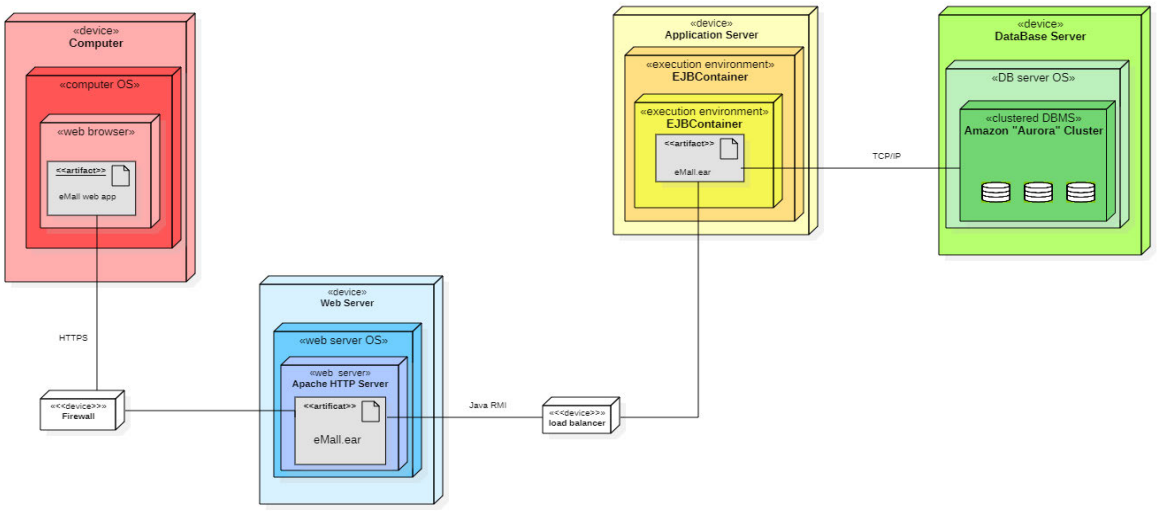


Figure 2.8 - CPMS Deployment view

Further details about the elements in the graph are provided in the following.

- **Web Server**
  The web server receives all the requests sent from CPO accessing the eMall web application via a web browser. It then forwards the received request to the application server for processing, by means of Java RMIs. Moreover, once the application server returns the results of the required computations, the web server generates the web page containing such data and forwards it to the CPO. In alternative, if the received request from the CPO asks for a static web page (e.g. the login form) , the web server returns it right away without even annoying the application server.

- **Load Balancer**
  A load balancer is a device which performs the load balancing. This is the process of distributing a set of requests over a group of resources such as application servers, web servers or else database servers, with the intent of increasing performance, scalability and soundness of the system.

- **Computer**
  A computer is a regular computer used by the CPO. No special applications are required to be installed on it, except for a web browser. This will be necessary to surf the internet and reach the eMall web page. An HTTPS connection with the system is established by passing through a firewall and after the forwarding, operated by the designated load balancer, to an available web server.

- **Database Server**
  The database server represents a cluster of databases managed via the MariaDB Galera Cluster. This will guarantee virtually synchronous data replication on all the members of the cluster. Therefore, even if the system can be slowed down by the complexity of the replication mechanism, it will grant a very high availability and always provide up-to-date information to the application server. In this way no errors are possible due, eventually, to the unalignment between master and slaves typical of semi-sync/async clusters.
- **Application server**
  The application server receives all the requests sent from smartphones which have the eMall application installed, QR-Readers and web servers. It is the most important element of the business tier, in fact it is designated for data elaboration and management. It is the only element of the architecture that can communicate with the database and it does so via TCP/IP.

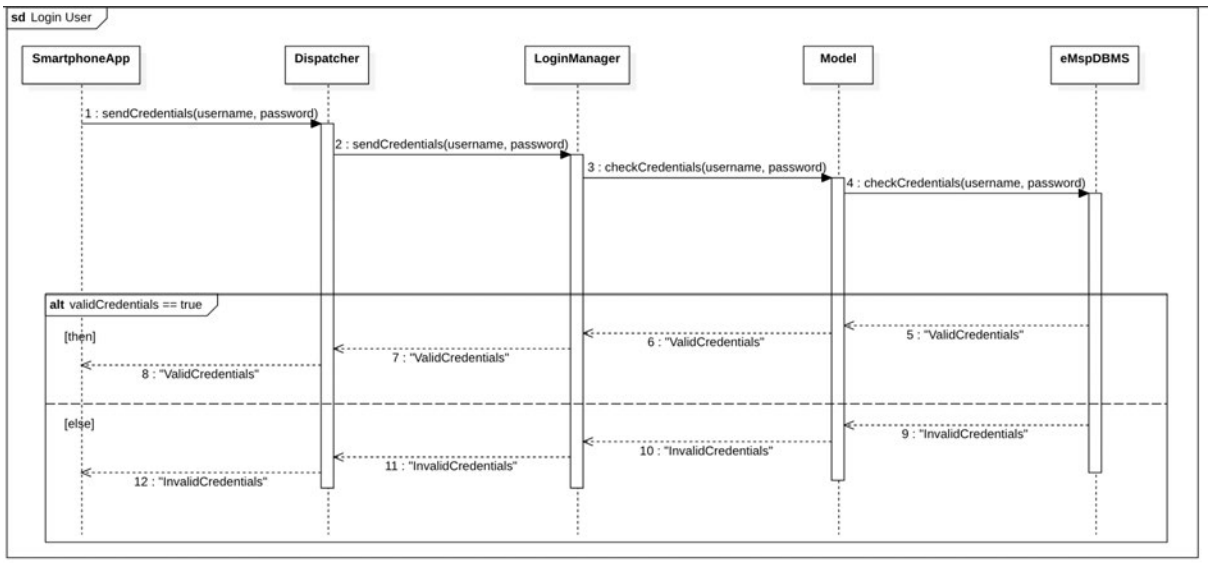
- **Firewall**
  The firewall is a device that monitors the packets incoming to the system, if a packet is potentially dangerous it is not forwarder. They are placed before the two load balancers, in this way the only packets that enter the network are considered safe, hence a DMZ is created in the net containing all the elements not in red of Figure 2.5.

- **Smartphone**
  The smartphone is a client on which it is possible to install the eMall smartphone application. The software can interact with the system via HTTPS requests, which are not handled by any web server but are forwarded directly to the application server, because they are already well-formatted and do not need any elaboration before being accepted by the application server.

2.4 Runtime views

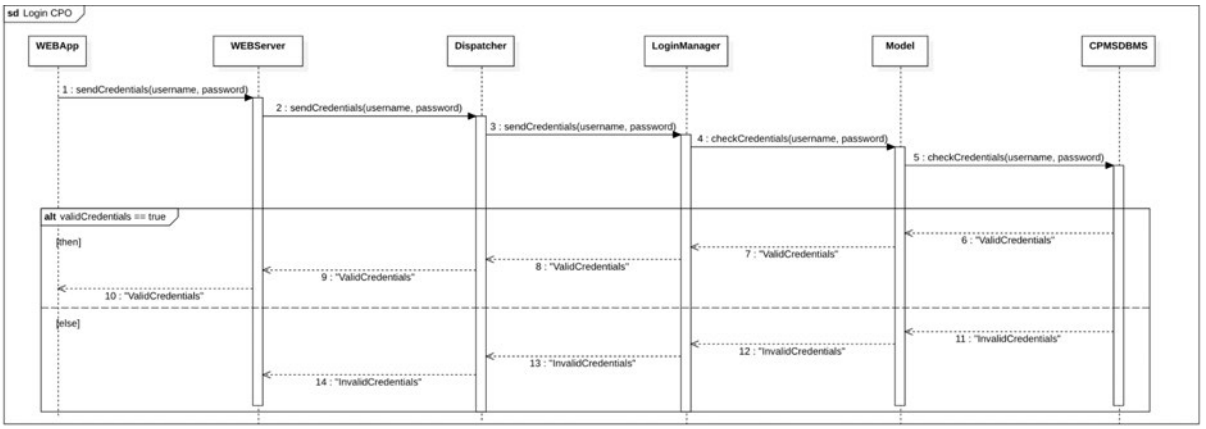
In this chapter are presented all the runtime views associated with the use cases described in the RASD relative to eMall. Runtime views show the interactions between the various components to carry out the functionalities offered by eMall.

Login User



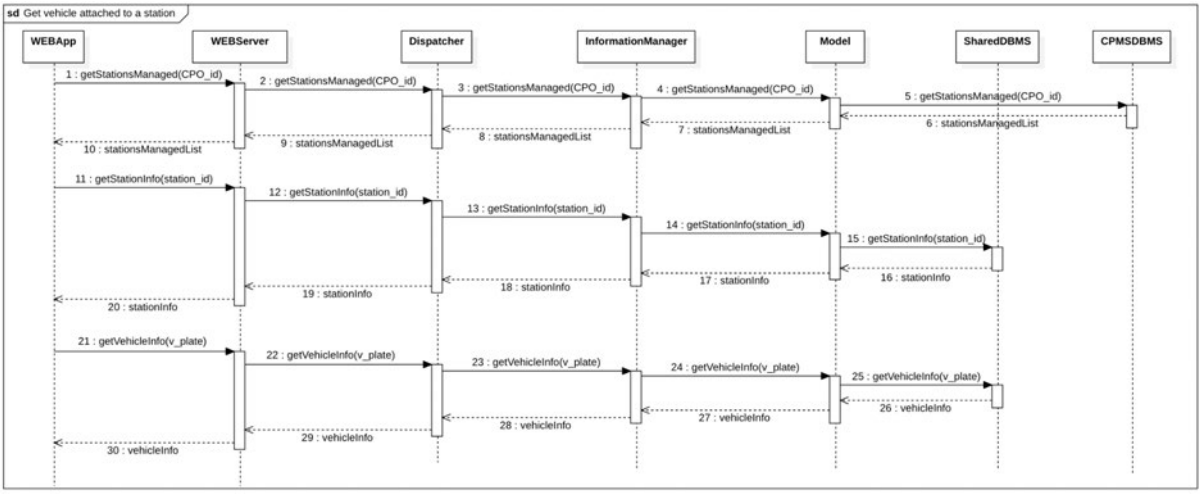
This sequence diagram represents the interaction that happens when a user wants to login to eMall application. The SmartphoneApp sends the user's credentials to the LoginManager component through the dispatcher. Then the LoginManager sends the credentials to the eMspDBMS through the model so that they can be checked.
At the end a message of valid or invalid credentials is sent back to the SmartphoneApp.

Login CPO



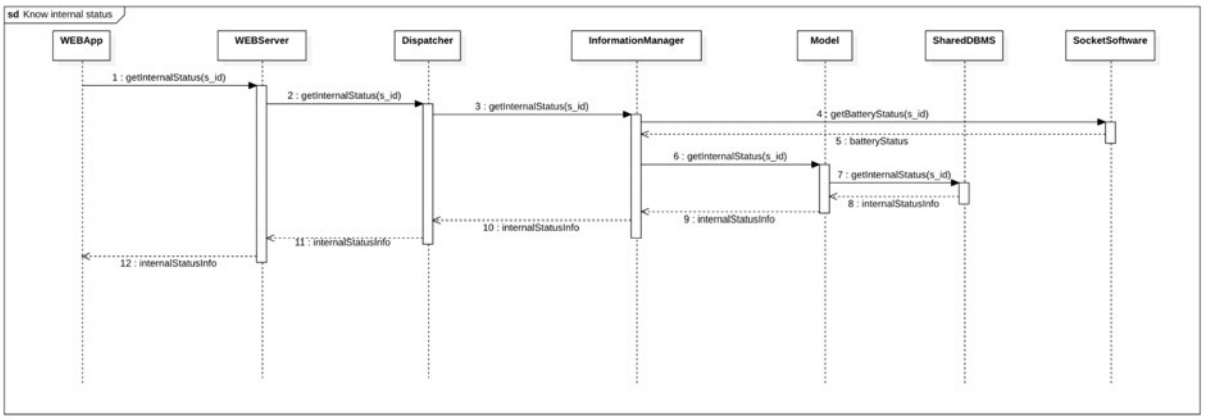
This sequence diagram represents the interaction that happens when a CPO wants to login to eMall application. The WEBApp sends the user's credentials to the LoginManager component through the WEBServer and the dispatcher. Then the LoginManager sends the credentials to the CPMSDBMS through the model so that they can be checked.
At the end a message of valid or invalid credentials is sent back to the WEBApp.

Get vehicle attached to a station



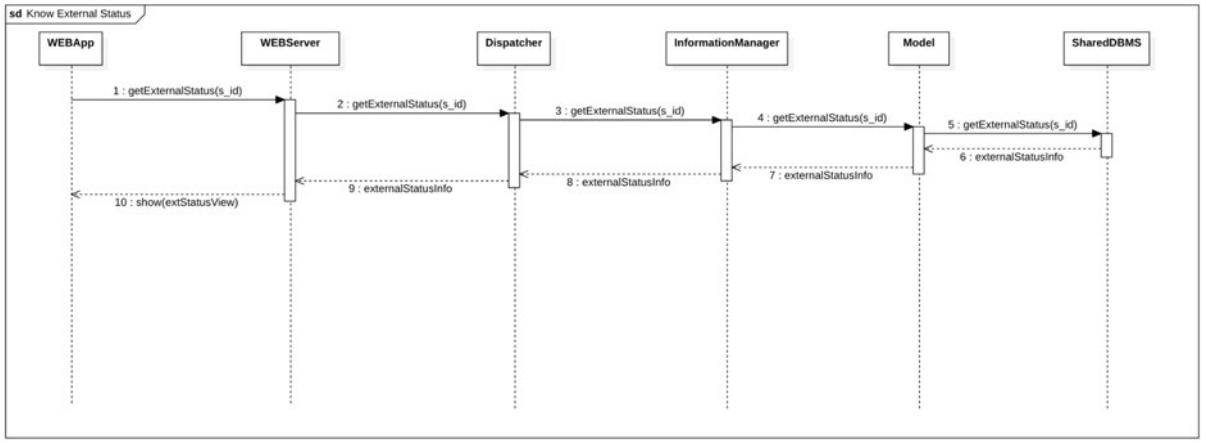
This sequence diagram represents the process of getting information about a vehicle attached to a station by a CPO. The WEBApp gets the stations managed by the CPO to show them through a query to the CPMSDBMS. Then the CPO selects one of the stations managed to see its information (and also the vehicles attached to its sockets), so the WEBApp gets the information of the station through a query to the SharedDBMS.
At the end the CPO selects one of the vehicles attached to the selected station and the WEBApp gets the information of the vehicle through a query to the SharedDBMS.
All the queries to the DBMSs go through the WEBServer, dispatcher, InformationManager and Model.

Know internal status



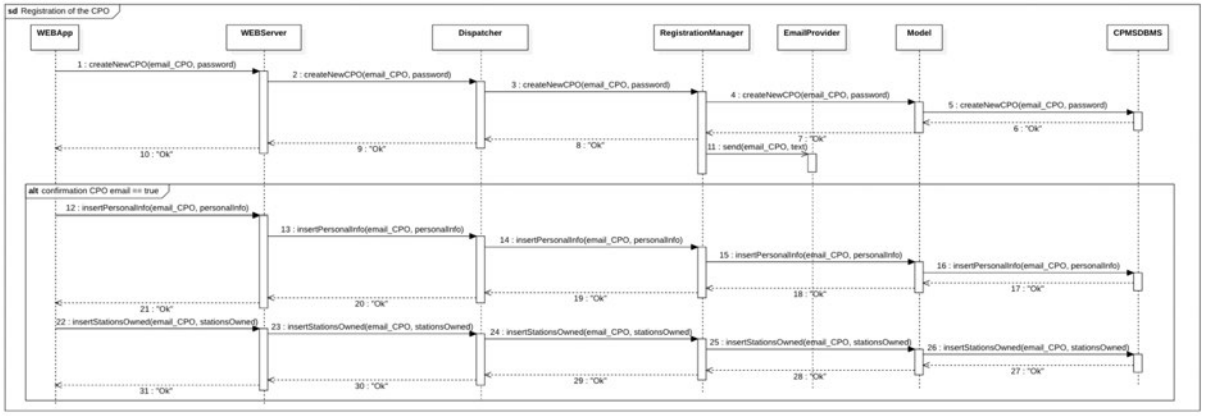
This sequence diagram represents the process through which a CPO gets a view of the internal status of a station. When the CPO has selected a station the WebAPP through WEBServer and then the Dispatcher sends the request to get the internal status of the selected station to the InformationManager. Then the InformationManager gets the battery status from the SocketSoftware of the selected station and then gets the other internal status information from the SharedDBMS. At the end the whole information is sent back to the WEBApp to be shown to the CPO.

Know external status



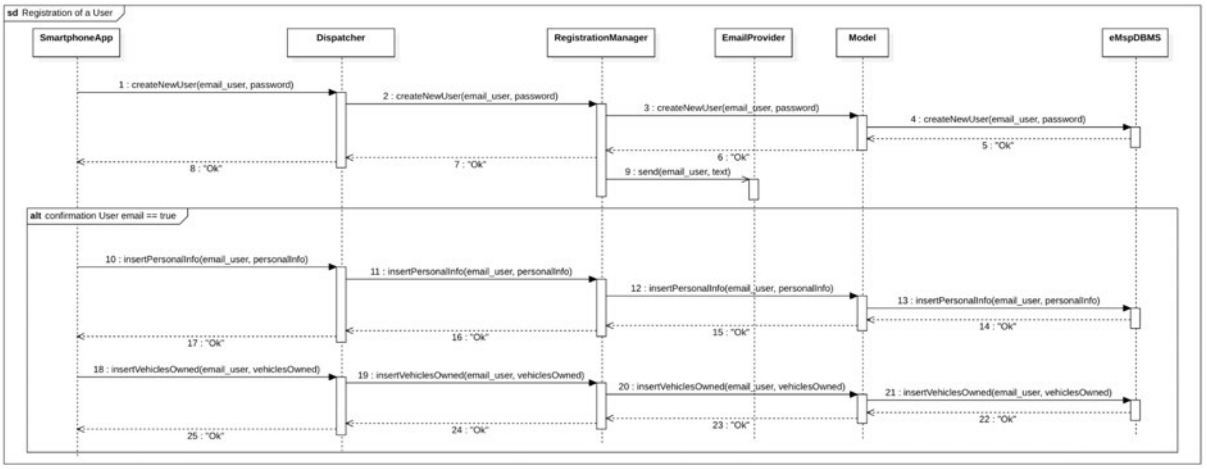
This sequence diagram represents the process through which a CPO gets a view of the external status of a station. When the CPO has selected a station the WebAPP through WEBServer and then the Dispatcher sends the request to get the external status of the selected station to the InformationManager. Then the InformationManager gets the internal status information from the SharedDBMS the information is sent back to the WEBApp to be shown to the CPO.

Registration of the CPO



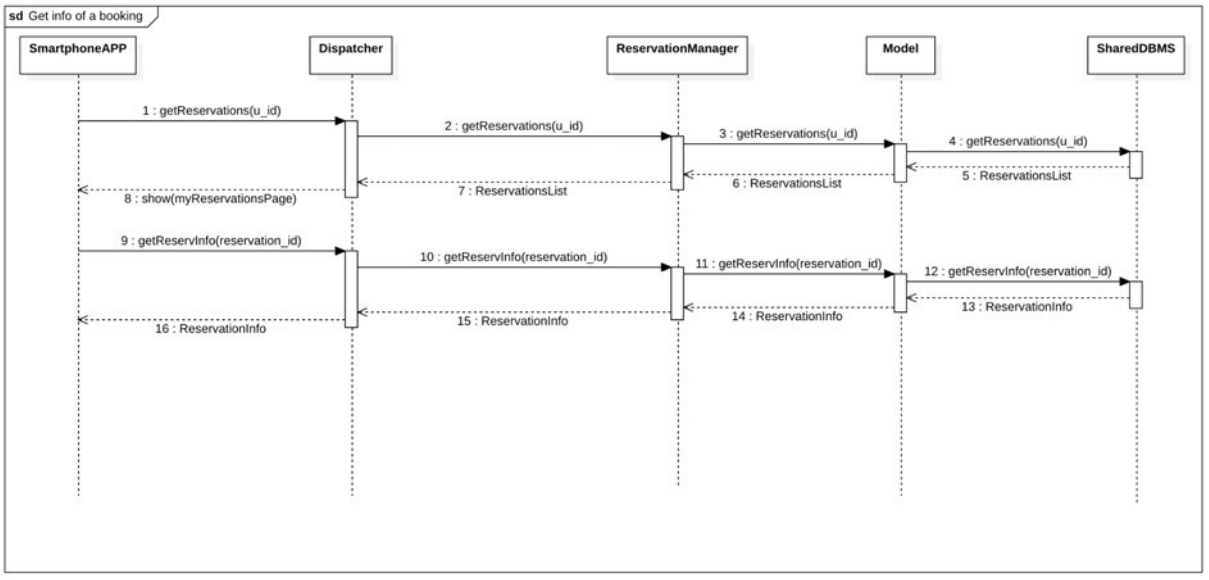
This sequence diagram shows the registration procedure followed by a CPO. The CPO inserts email and password for the registration and the WEBApp sends these data to the CPODBMS through the RegistrationManager component.
Then the RegistrationManager sends an email to the CPO containing the link to confirm the email by using the EmailProvider.
After the email confirmation by the CPO, the CPO inserts his personal info and then he inserts the stations that he manages.
All the data sent from the WEBApp to the CPODBMS pass through the WEBServer, Dispatcher, RegistrationManager and Model.

Registration of a User



This sequence diagram shows the registration procedure followed by a user. The user inserts email and password for the registration and the SmartphoneApp sends these data to the eMspDBMS through the RegistrationManager component.
Then the RegistrationManager sends an email to the user containing the link to confirm the email by using the EmailProvider.
After the email confirmation by the user, the user inserts his personal info and then he inserts the vehicles that he owns.
All the data sent from the SmartphoneApp to the eMspDBMS pass through the Dispatcher, RegistrationManager and Model.

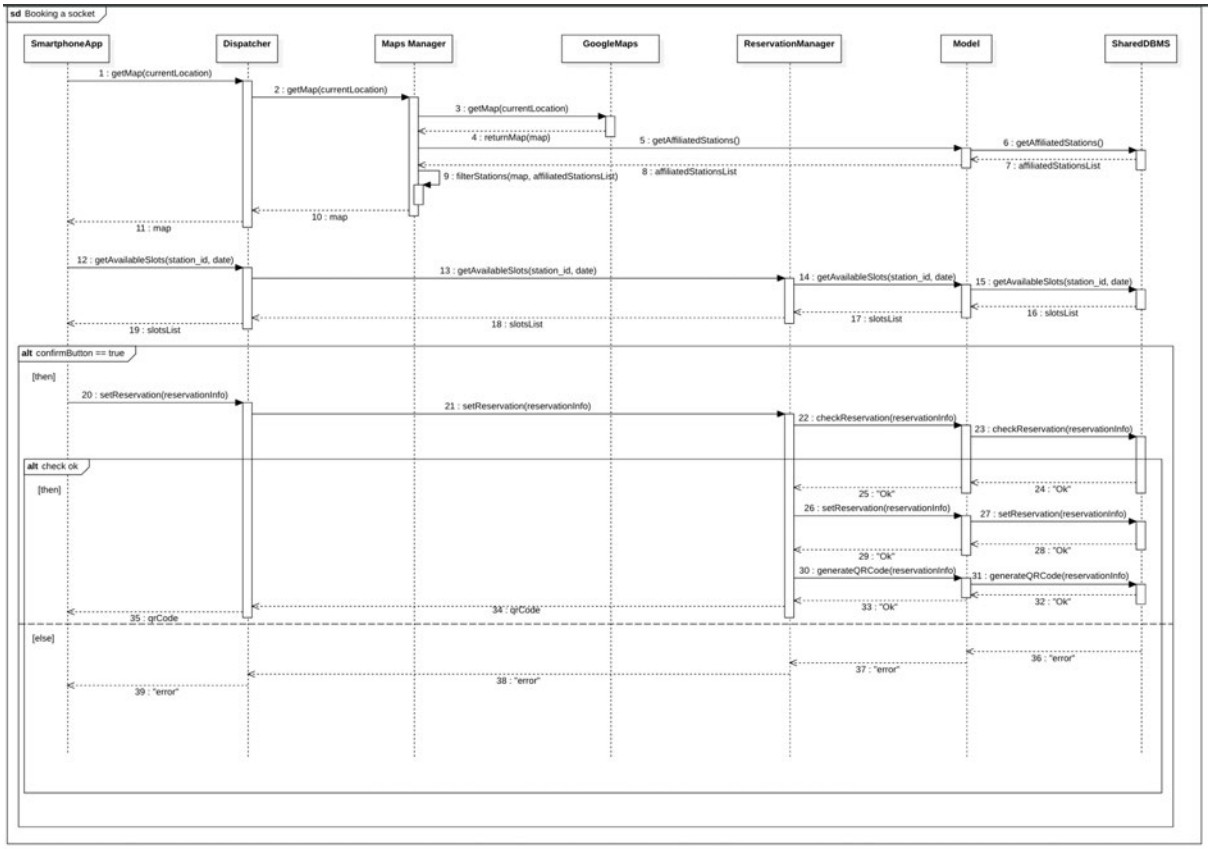
Get info of a booking



This sequence diagram shows the procedure of a user that wants to see the information about one of his bookings. To get the reservations to be shown on the MyReservation page the SmartphoneApp sends a request to the SharedDBMS through the ReservationManager component. Then after getting the user's reservation the MyReservation page is loaded with the list of the reservations and when the user selects one of them a new request is sent from the SmartphoneApp to the SharedDBMS through the ReservationManager.

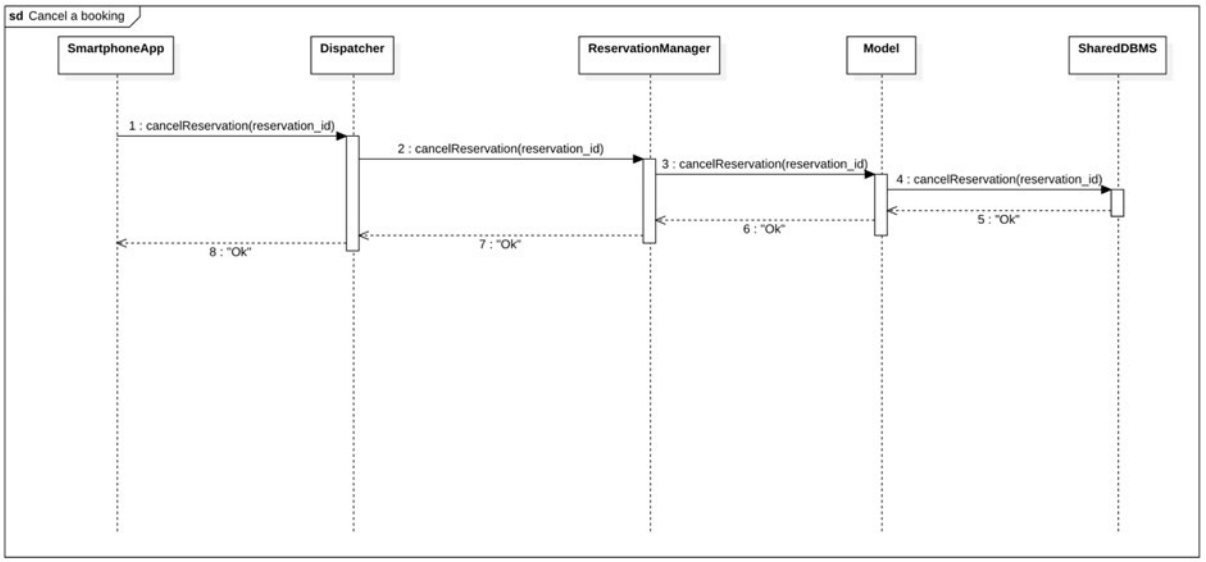
Then is loaded a page with the info of the selected reservation got by the SharedDBMS.
All the requests from the SmartphoneApp to the Shared DBMS pass through the Dispatcher, ReservationManager and then Model.

Booking a socket



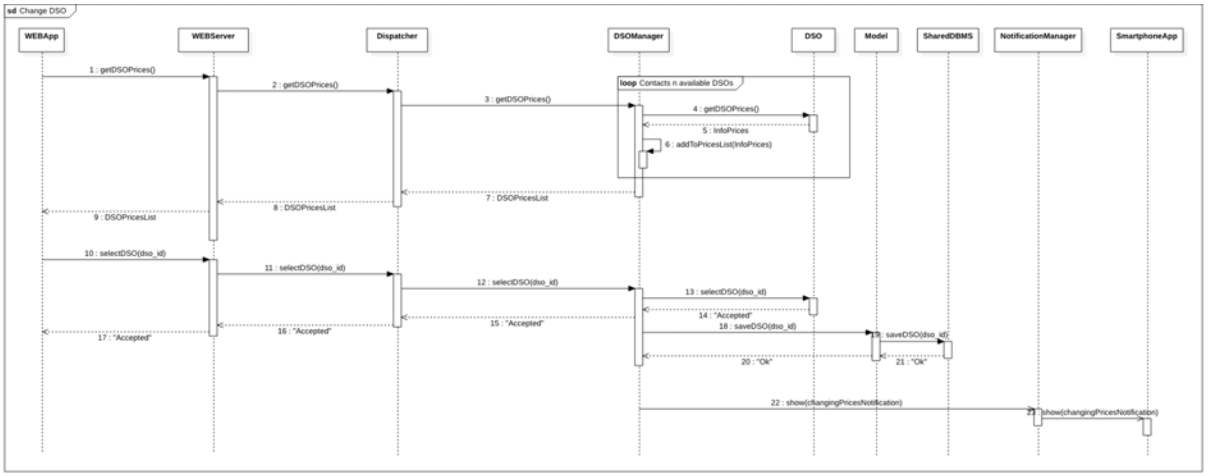
This sequence diagram represents the procedure of a user that books a socket for recharging. The SmartphoneApp sends a request to GoogleMaps through the MapsManager component to get a map based on the currentLocation of the user. Then the MapsManager queries the SharedDBMS through the ReservationManager to get the list of the stations that are affiliated to eMall and then it filters the stations of the map with the only ones affiliated to eMall and sends the filtered map back to the SmartphoneApp.
Later the SmartphoneApp, after the user has selected a station and a date from the map, sends a query to the SharedDBMS to get the available slots for the booking.
Then, after the user has selected one or more slots and he has pressed the confirm button, the SmartphoneApp sends a query to set the booking to the ReservationManager; the ReservationManager, before setting the reservation, checks through the SharedDBMS if the user has already done other reservations during the selected timeframe. If the check is ok then the reservation is setted in the SharedDBMS and the ReservationManager generates the QR-Code, saves it into the SharedDBMS and sends it back to the SmartphoneApp. If instead the check isn't ok the SharedDBMS sends back an error message until the SmartphoneApp.

Cancel a booking



This sequence diagram represents the procedure of canceling a booking by a user. The user selects a reservation he wants to cancel and a request to cancel the reservation is sent from the SmartphoneApp to the SharedDBMS through the Dispatcher, the ReservationManager and the Model.

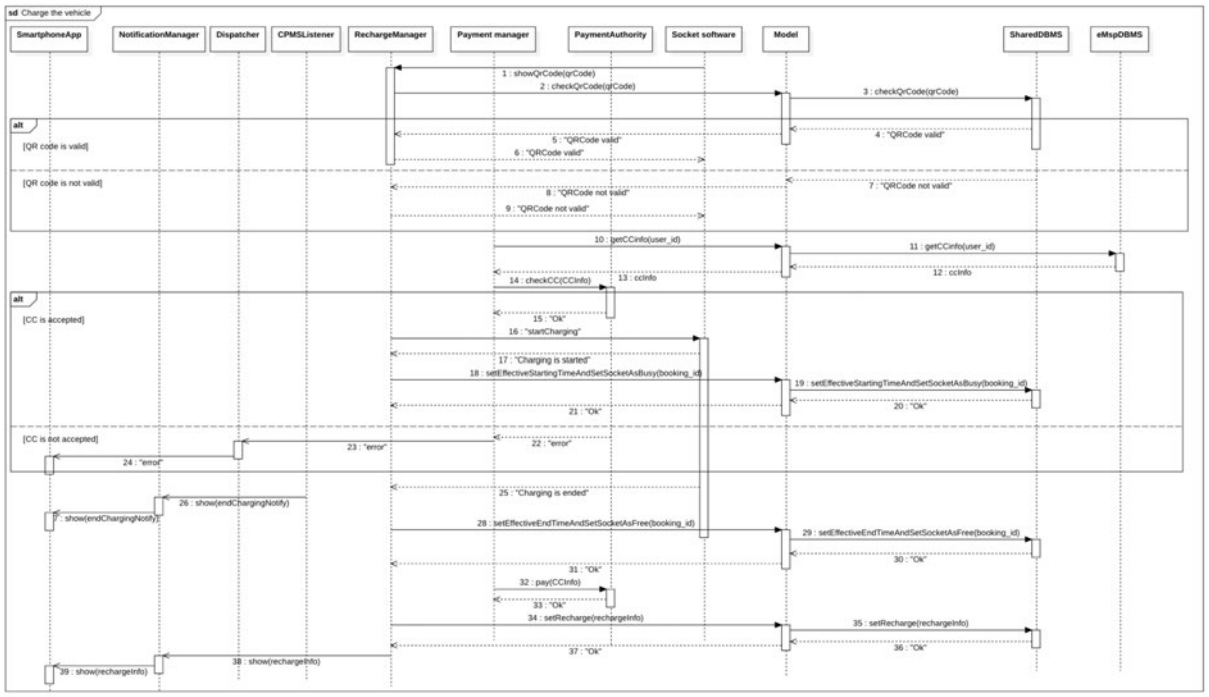
Change DSO



This sequence diagram represents the process of changing a DSO from which to acquire energy by a CPO for the stations he manages. The WEBApp sends a request to the DSOManager through the WEBServer and Dispatcher to get the prices of the available DSOs. Then the DSOManager sends the request to the DSOs and every time he gets the price of a DSO he adds it to a list and when it has contacted every DSO it sends back the list containing all the prices back to the WEBApp.
Then the DSO selects a DSO from which to acquire energy and the WEBApp sends a request of selectDSO to the DSO through the DSOManager. The DSOManager sends the request to the DSO and saves the choice into the SharedDBMS through the Model.

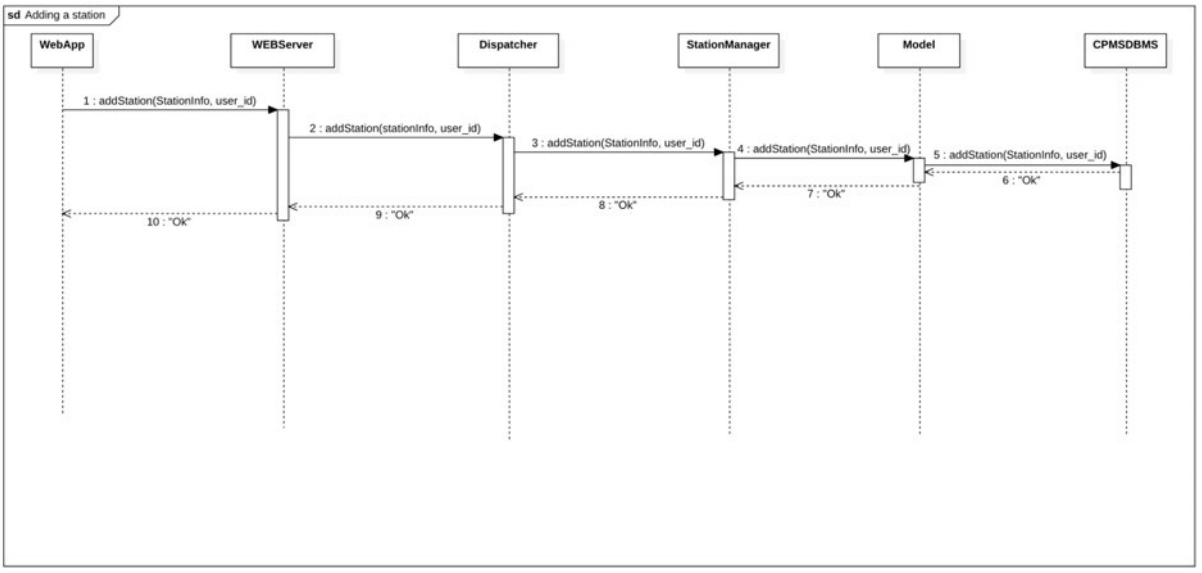
At the end the DSOManager uses the NotificationManager to send a notification of changing prices to the SmartphoneApp of the users that had a booking on the stations involved.

Charge the vehicle



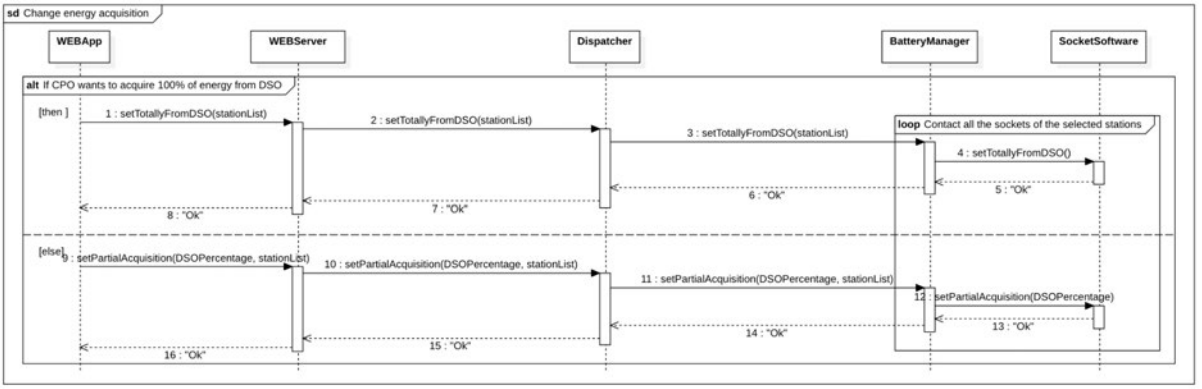
This sequence diagram describes the process of charging a user's vehicle. The SocketSoftware sends the QRcode (shown by the user) to the RechargeManager and the RechargeManager checks it by doing a query to the SharedDBMS. If the QRcode is valid the Rechargemanager tells the PaymentManager to start checking the user's cc card by contacting the payment authority (after having got cc info by the eMspDBMS). If the cc is not accepted an error message is sent back, but if it is accepted the PaymentManager tells the RechargeManager to start the charging by contacting the SocketSoftware. Then the RechargeManager contacts the Model and the SharedDBMS to set the effective starting time of the recharge and to set the socket as busy. Once the charging ends the SocketSoftware tells it to the RechargeManager and the RechargeManager tells it to the CPMSListener so that it can send to the user's smartphone a notification of end charging.
In the meanwhile the RechargeManager sets the effective ending time and sets the socket as free through the Model and the SharedDBMS.
Later the RechargeManager, after having told the PaymentManager to process the payment by the PaymentAuthority, sets the recharge (with all its info) on the SharedDBMS and contacts the NotificationManager to send a notification of recharge recap to the SmartphoneApp.
All the communication between RechargeManager and PaymentManager and also between the RechargeManager and the CPMSListener occurs through the OCPI protocol so it is not represented in the sequence diagram.

Adding a station



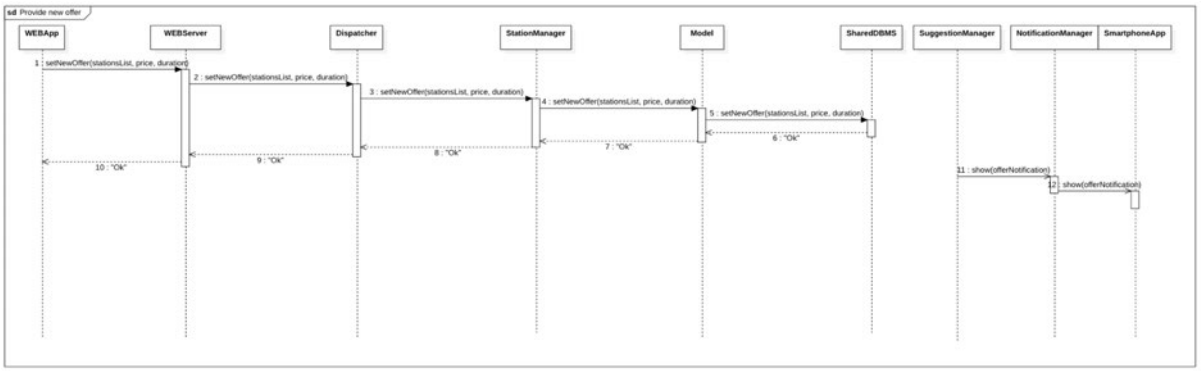
This sequence diagram represents the process of adding a station to those managed by a CPO. The WEBApp sends a query to add a station to those managed to the CPMSDBMS through WebServer, Dispatcher, StationManager, Model.

Change energy acquisition



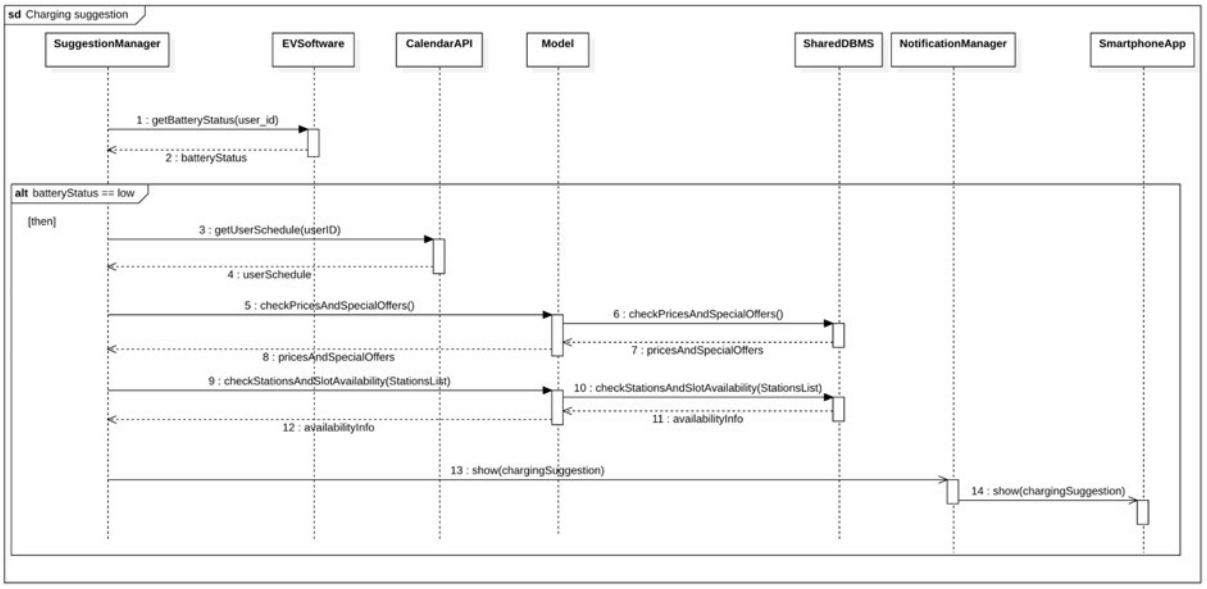
This sequence diagram represents the procedure of changing the way to acquire energy by the CPO. The WEBApp sends a request to the SocketSoftware of the selected stations to set the total or partial acquisition of energy from the DSO. The request goes from the WEBApp to the SocketSoftware by going through the WebServer, Dispatcher, BatteryManager.

Provide new offer



This sequence diagram represents the creation of a new offer by a CPO. The WEBApp sets the new offer for the selected stations by doing a query to the shared DBMS through the StationManager component. Then the user will be notified about the offer by the SuggestionManager.

Charging suggestion



This sequence diagram represents the smart suggestions by the system to the user's smartphone. The SuggestionManager gets the status of the battery from the EVSoftware and if the battery is low it gets the user's schedule by the CalendarAPI. Then the SuggestionManager gets prices and special offers and also stations and their availability most suitable for the user's schedule by doing queries to the SharedDBMS. At the end the SuggestionManager sends a notification to the SmartphoneApp through the NotificationManager

2.5 Component interfaces

This section lists all the methods that each component interface provides to the other components.

- **DispatcherI**
  - sendCredentials(username, password)
  - getStationsManaged(CPO_id)
  - getStationInfo(station_id)
  - getVehicleInfo(v_plate)
  - getInternalStatus(s_id)
  - getExternalStatus(s_id)
  - createNewCPO(email_CPO, password)
  - insertPersonalInfo(email_CPO, personalInfo)
  - insertStationsOwned(email_CPO, stationsOwned)
  - createNewUser(email_user, password)
  - insertPersonalInfo(email_user, personalInfo)
  - insertVehiclesOwned(email_user, vehiclesOwned)
  - getReservations(u_id)
  - getReservInfo(reservation_id)
  - getMap(currentLocation)
  - getAvailableSlots(station_id, date)
  - setReservation(reservationInfo)
  - cancelReservation(reservation_id)
  - getDSOPrices()
  - selectDSO(dso_id)
  - addStation(StationInfo, user_id)
  - setTotallyFromDSO(stationList)
  - setPartialAcquisition(DSOPercentage, stationList)
  - setNewOffer(stationsList, price, duration)

- **LoginManagerI**
  - sendCredentials(username, password)

- **ModelI**
  - checkCredentials(username, password)
  - getStationsManaged(CPO_id)
  - getStationInfo(station_id)
  - getVehicleInfo(v_plate)
  - getInternalStatus(s_id)
  - getExternalStatus(s_id)
  - createNewCPO(email_CPO, password)
  - insertPersonalInfo(email_CPO, personalInfo)

- insertStationsOwned(email_CPO, stationsOwned)
- createNewUser(email_user, password)
- insertPersonalInfo(email_user, personalInfo)
- insertVehiclesOwned(email_user, vehiclesOwned)
- getReservations(u_id)
- getReservInfo(reservation_id)
- getAffiliatedStations()
- getAvailableSlots(station_id, date)
- setReservation(reservationInfo)
- checkReservation(reservationInfo)
- generateQRCode(reservationInfo)
- cancelReservation(reservation_id)
- saveDSO(dso_id)
- checkQrCode(qrCode)
- getCCinfo(user_id)
- setEffectiveStartingTimeAndSetSocketAsBusy(booking_id)
- setEffectiveEndTimeAndSetSocketAsFree(booking_id)
- setRecharge(rechargeInfo)
- addStation(StationInfo, user_id)
- setNewOffer(stationsList, price, duration)
- checkPricesAndSpecialOffers()
- checkStationsAndSlotAvailability(StationsList)

- **eMspDBMSI**
  - checkCredentials(username, password)
  - createNewUser(email_user, password)
  - insertPersonalInfo(email_user, personalInfo)
  - insertVehiclesOwned(email_user, vehiclesOwned)
  - getCCinfo(user_id)

- **WEBServerI**
  - sendCredentials(username, password)
  - getStationsManaged(CPO_id)
  - getStationInfo(station_id)
  - getVehicleInfo(v_plate)
  - getInternalStatus(s_id)
  - getExternalStatus(s_id)
  - createNewCPO(email_CPO, password)
  - insertPersonalInfo(email_CPO, personalInfo)
  - insertStationsOwned(email_CPO, stationsOwned)
  - getDSOPrices()
  - selectDSO(dso_id)

- addStation(StationInfo, user_id)
- setTotallyFromDSO(stationList)
- setPartialAcquisition(DSOPercentage, stationList)
- setNewOffer(stationsList, price, duration)

- **CPMSDBMSI**
  - checkCredentials(username, password)
  - getStationsManaged(CPO_id)
  - createNewCPO(email_CPO, password)
  - insertPersonalInfo(email_CPO, personalInfo)
  - insertStationsOwned(email_CPO, stationsOwned)
  - addStation(StationInfo, user_id)

- **InformationManagerI**
  - getStationsManaged(CPO_id)
  - getStationInfo(station_id)
  - getVehicleInfo(v_plate)
  - getInternalStatus(s_id)
  - getExternalStatus(s_id)

- **SharedDBMSI**
  - getStationInfo(station_id)
  - getVehicleInfo(v_plate)
  - getInternalStatus(s_id)
  - getExternalStatus(s_id)
  - getReservations(u_id)
  - getReservInfo(reservation_id)
  - getAffiliatedStations()
  - getAvailableSlots(station_id, date)
  - setReservation(reservationInfo)
  - checkReservation(reservationInfo)
  - generateQRCode(reservationInfo)
  - cancelReservation(reservation_id)
  - saveDSO(dso_id)
  - checkQrCode(qrCode)
  - setEffectiveStartingTimeAndSetSocketAsBusy(booking_id)
  - setEffectiveEndTimeAndSetSocketAsFree(booking_id)
  - setRecharge(rechargeInfo)
  - setNewOffer(stationsList, price, duration)
  - checkPricesAndSpecialOffers()
  - checkStationsAndSlotAvailability(StationsList)

- **SocketSoftwareI**
  - getBatteryStatus(s_id)
  - setTotallyFromDSO()
  - setPartialAcquisition(DSOPercentage)

- **RegistrationManagerI**
  - createNewCPO(email_CPO, password)
  - insertPersonalInfo(email_CPO, personalInfo)
  - insertStationsOwned(email_CPO, stationsOwned)
  - createNewUser(email_user, password)
  - insertPersonalInfo(email_user, personalInfo)
  - insertVehiclesOwned(email_user, vehiclesOwned)

- **EmailProviderI**
  - send(email_CPO, text)
  - send(email_user, text)

- **ReservationManagerI**
  - getReservations(u_id)
  - getReservInfo(reservation_id)
  - getAvailableSlots(station_id, date)
  - setReservation(reservationInfo)
  - cancelReservation(reservation_id)

- **MapsManagerI**
  - getMap(currentLocation)

- **GoogleMapsI**
  - getMap(currentLocation)

- **DSOManagerI**
  - getDSOPrices()
  - selectDSO(dso_id)

- **DSOI**
  - getDSOPrices()
  - selectDSO(dso_id)

- **NotificationManagerI**
  - show(changingPricesNotification)
  - show(endChargingNotify)
  - show(rechargeInfo)

- show(offerNotification)
- show(chargingSuggestion)

- **NotificationsI**
  - show(changingPricesNotification)
  - show(endChargingNotify)
  - show(rechargeInfo)
  - show(offerNotification)
  - show(chargingSuggestion)

- **RechargeManagerI**
  - showQrCode(qrCode)

- **PaymentAuthorityI**
  - checkCC(CCInfo)
  - pay(CCInfo)

- **StationManagerI**
  - addStation(StationInfo, user_id)
  - setNewOffer(stationsList, price, duration)

- **BatteryManagerI**
  - setTotallyFromDSO(stationList)
  - setPartialAcquisition(DSOPercentage, stationList)

- **EVSoftwareI**
  - getBatteryStatus(user_id)

- **CalendarAPII**
  - getUserSchedule(userID)

2.6 Selected architectural Styles and patterns

2.6.1 3-tier Architecture

eMall will be built over a 3-tier architecture which provides many benefits thanks to the modularization of the system in three independent layers (or tiers):

1. **Presentation tier**: Top-level tier including the customers interface. Its main function is to rearrange the received data from the application tier and present them to the customers in a more intuitive and comprehensible manner.
2. **Business Logic tier**: It includes the business logic of the application, that is the logic according to which the application takes decisions and performs calculations. Moreover, it passes and processes data between the two surrounding layers.
3. **Data tier**: It includes the Database system and provides an API, to the application tier, for accessing and managing the data stored in the DB

Adopting this type of architecture guarantees a higher flexibility by allowing at first to develop and then to update a specific part of the system apart from the others. Besides, a middle tier between client and data server ensures a better protection of the stored data. In fact the information is accessed through the application layer instead of being accessed directly by the client.

2.6.2 Model View Controller pattern

For implementing the web application, that allows the user and the operator to access all the functionalities offered by eMall, was chosen to follow the Model-View-Controller pattern (MVC pattern). It includes three main components:

- **Model**: contains the application's data and provides methods for its manipulation
- **View:** it contains all the different possible visual representations of the data
- **Controller**: sits between the model and the view. Upon the occurrence of events (for example the pression of a button) execute a predefined reaction which may include some operations on the model that are then reflected on the view

The distribution of tasks between these three elements translates into an increased decoupling of the components which leads to a series of benefits such as reusability, simplicity of implementation, etc

2.6.3 Facade pattern

The facade pattern is used in the implementation of the Dispatcher component to hide the complexity of the application server to the client applications to which is provided a simpler interface. This solution ulteriorly increases the decoupling of the various components of our system; in particular between the client applications and application servers.

2.6.4 Mediator pattern

The mediator pattern is a behavioral pattern used to reduce coupling between modules willing to communicate with each other. The mediator component sits in between two or more objects and encapsulates how such objects communicate; this way not requiring them to know implementation details about each other. In our system there are several mediator such:

- **MapsManager** acts as a mediator linking the internal components of the application server with the external maps provider. This solution guarantees a greater flexibility of the system which can adapt to a change of the maps provider by simply modifying the MapsManager.
- **PaymentManager** acts as a mediator linking the internal components of the application server with the external payment provider. This solution guarantees a greater flexibility of the system which can adapt to a change of the payment provider by simply modifying the PaymentManager.
- **SocketSoftwareManager**, **BatteryManager, InformationManager** and **RechargeManager** act as mediators linking the internal components of the application server with the external SocketSoftware provider. This solution guarantees a greater flexibility of the system which can adapt to a change of the payment provider by simply modifying the SocketSoftwareManager.
- **DSOManager** acts as a mediator linking the internal components of the application server with the external DSO provider. This solution guarantees a greater flexibility of the system which can adapt to a change of the payment provider by simply modifying the DSOManager.
- **RegistrationManager** acts as a mediator linking the internal components of the application server with the external Email provider. This solution guarantees a greater flexibility of the system which can adapt to a change of the email provider by simply modifying the RegistrationManager.
- **SuggestionManager** acts as a mediator linking the internal components of the application server with the external EV software and API calendar provider. This solution guarantees a greater flexibility of the system which can adapt to a change of the EV software and API calendar provider by simply modifying the SuggestionManager.

2.7 Other design decisions

In this section are presented some of the design decisions made for the system to make it work as expected.

2.7.1 Availability

Load balancing and replication are concepts that have been introduced in the system to increase the availability and reliability of the system. In this way the system would be as fault-tolerant as possible regarding data management and service availability.

2.7.2 Notification timed

The task of managing notifications to users is assigned to the SmartphoneApplication component. It is important to recall the fact that the smartphone stores a copy of the booking, including the station's location, while confirming a booking. This information is necessary to the component because it will regularly check, as a background process, the user's reservation. When the starting time is reasonably close to the recharge's beginning time, then a notification is generated. This will move the computation to the SmartphoneApplication component, avoiding the ApplicationServer to check regularly in the database the booking in the "approaching" state, thus saving some computational resources.

2.7.1 Data Storage

Another important aspect that has been taken into account is the management of the different sub-system. There are two not shared DBs that are used respectively by the CPMS and eMsp where only relevant data related to that sub-system is stored. The shared DB is used to store data that are relevant for both sub-system so you do not have any duplicate data i.e data that are used by both such as information regarding the reservation.

3. User Interface Design

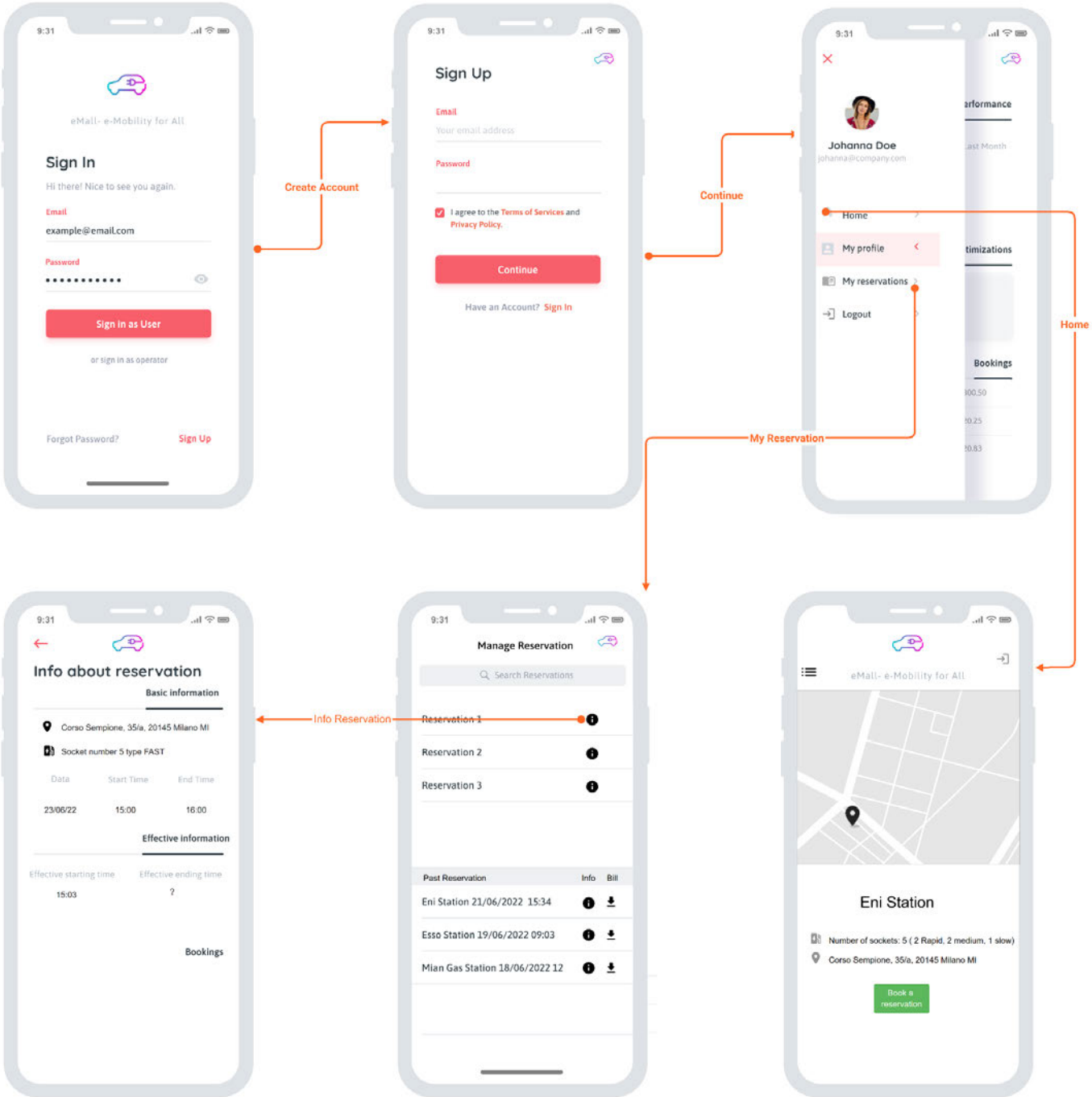


Figure 3.1 - Users functionalities

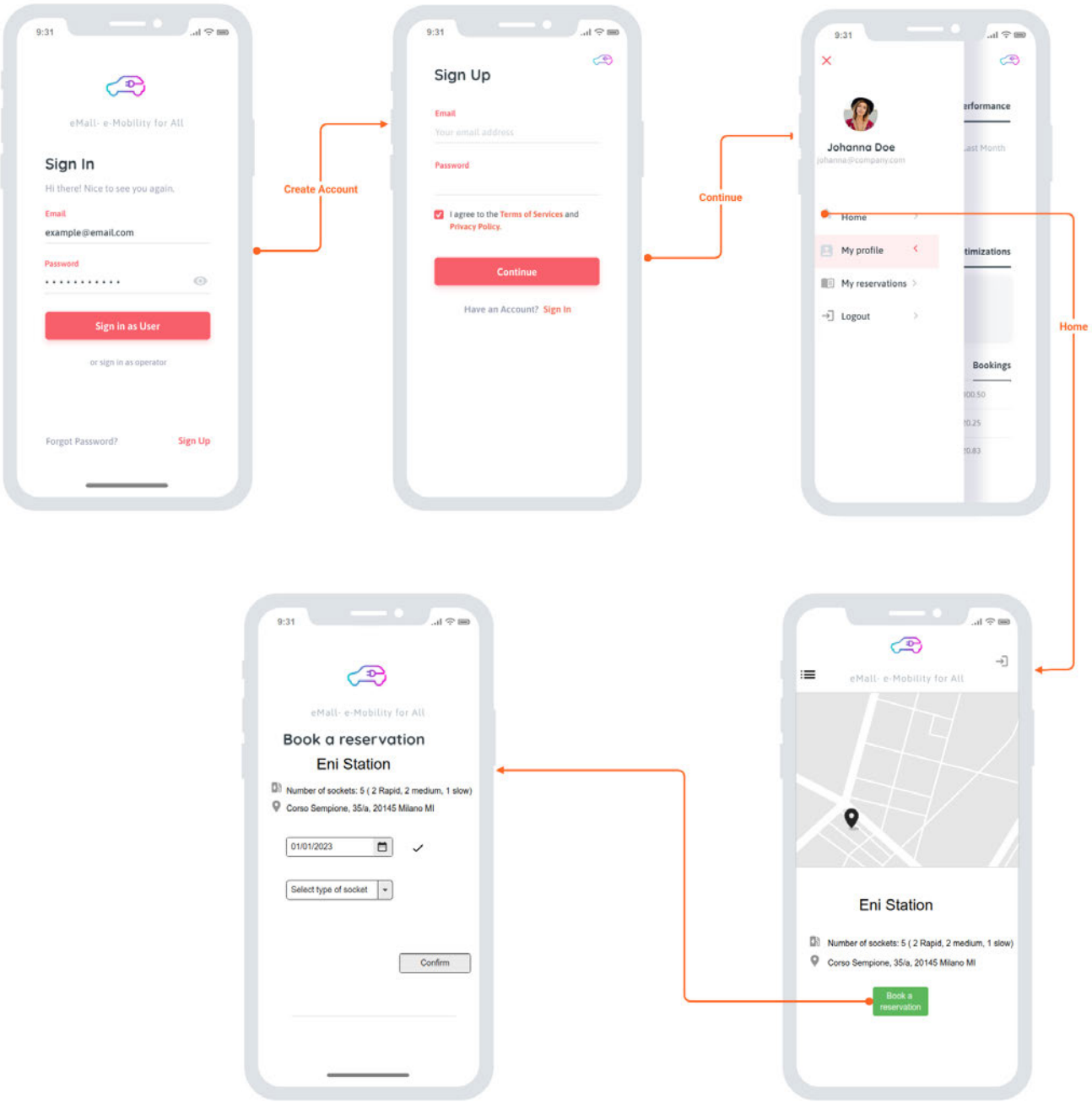


Figure 3.2 - Book a reservation functionality

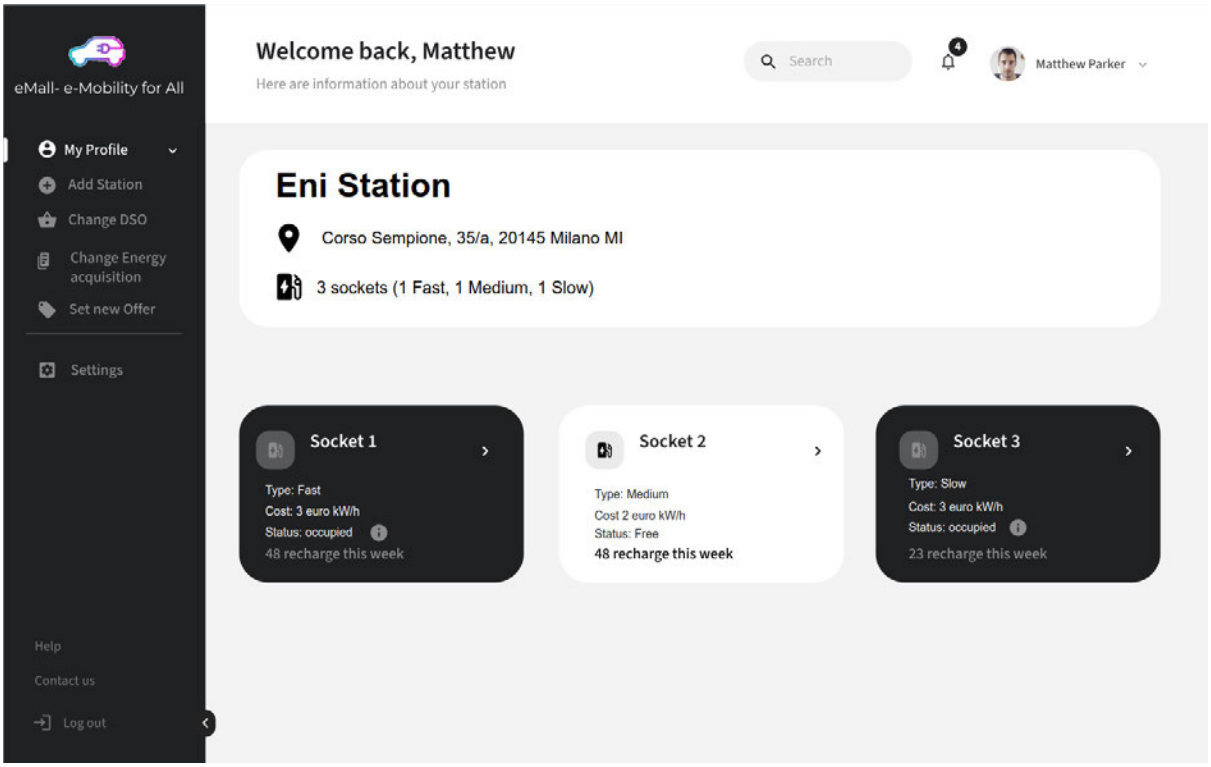


Figure 3.3 - Station Manager view

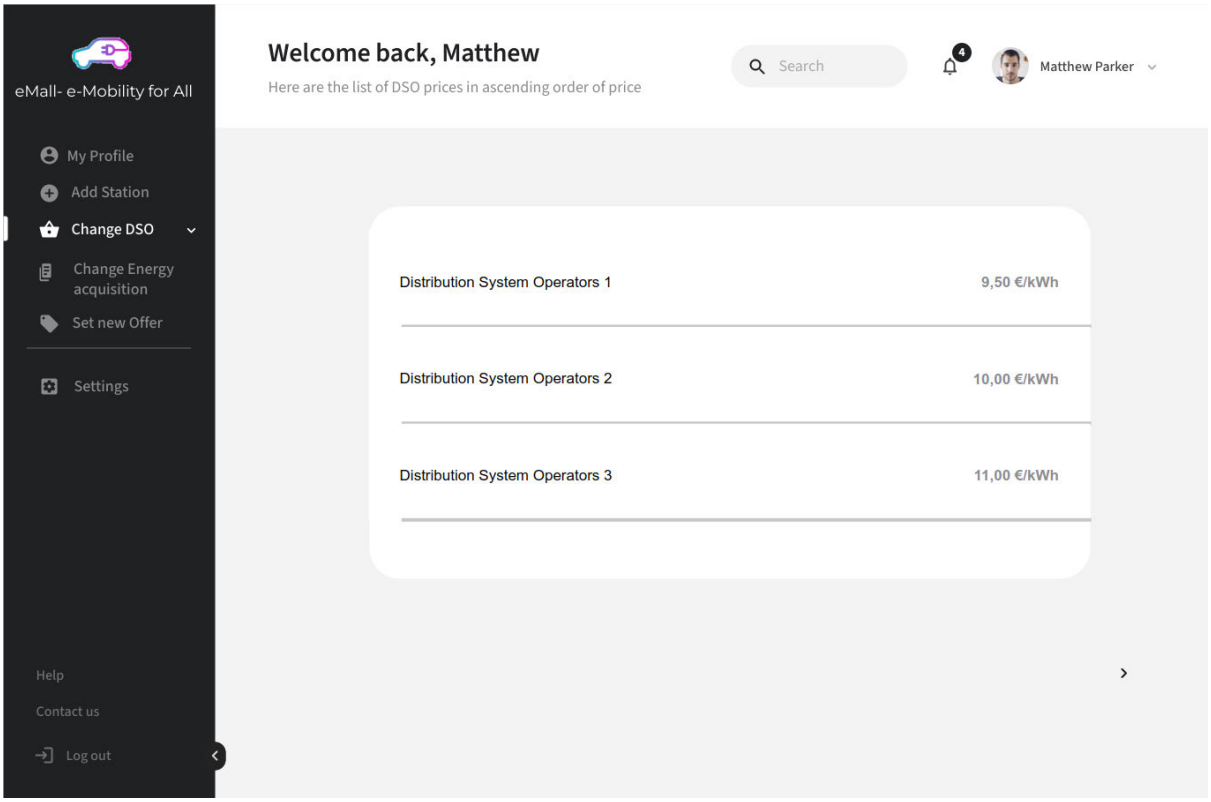


Figure 3.4 - DSO prices view

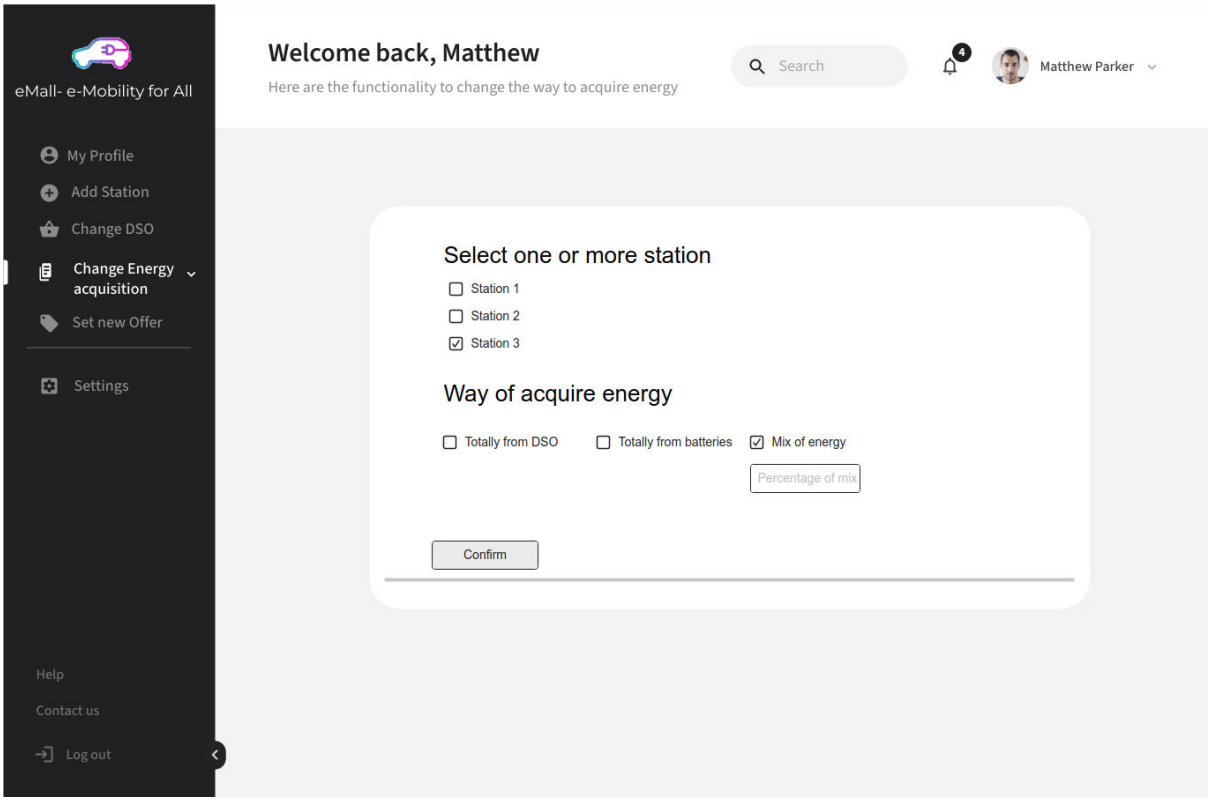


Figure 3.5 - Energy acquisition view

4. Requirement traceability

In this section of the document it is explained, for each requirement defined in the RASD, what design elements are involved for its fulfillment. In the following a series of tables maps such design elements to the respective requirement

Requirements:	[R2] The system allows CPOs to sign up
Components:	● WebApp ● WebServer ● Application server: ○ Dispatcher ○ RegistrationManager ○ Model ● DBMS

Requirements:	[R3] The system allows registered CPOs to login
Components:	● WebApp ● WebServer ● Application server: ○ Dispatcher ○ LoginManager ○ Model ● DBMS

Requirements:	[R4] The system allows registered users to login
Components:	● SmartphoneApp ● Application server: ○ Dispatcher ○ LoginManager ○ Model ● DBMS

Requirements:	[R5] The system allows CPOs to insert information about station for which they are responsible
Components:	● WebApp ● WebServer ● Application server: ○ Dispatcher ○ StationManager ○ Model ● DBMS

Requirements:	[R6] The system allows CPOs to see the list of stations that they manage [R7] The system allows CPOs to see the exact number of vehicle attached to a station that they manage [R8] The system allows CPOs to get the information about the recharged vehicle [R9] The system allows CPOs to see the details of a booking [R10] The system allows CPOs to see all the details of booking during a specific timeFrame
Components:	● WebApp ● WebServer ● ApplicationServer: ○ Dispatcher ○ InformationManager ■ StatusStationManager ■ StatisticsManager ○ Model ● DBMS

Requirements:	[R13] The system allows users to select a charging station(and so a socket) from a map [R14] The system allows users to search for a specific station [R15] The system allows users to select the exact date and time of the slot they are booking from a list of available timeFrame [R16] The systems allows users to cancel a booking [R17] The system allows users to see all the previous charging process occurred [R18] The system allows users to select the estimated duration of a charging process [R26] The system disallows user to take two different socket in the same time slot
Components:	● SmartphoneApp ● GoogleMaps (external component) ● ApplicationServer: ○ Dispatcher ○ ReservationManager ■ AvailabilityManager ■ ReservationMaker ○ MapsManager ○ Model ● DBMS

Requirements:	[R11] The system allows CPOs to modify the way in which socket batteries are charged [R12] The system allows CPOs to decide from which DSO acquire energy
Components:	● WebApp ● WebServer ● ApplicationServer: ○ Dispatcher ○ DSOManager ■ GetPriceManager ■ DSOSelectorManager ○ BatteryManager ○ Model ● DBMS

Requirements:	[R1]The system allows users to sign up [R19] The system allows users to insert their personal information (i.e. first name, second name,telephone number, email, and credit cards details)
Components:	● SmartphoneApp ● ApplicationServer: ○ Dispatcher ○ RegistrationManager ○ Model ● DBMS

Requirements:	[R21] Notify users that use the application when the time of their booking is approaching [R22] Notify users when the charging process ended or when time slot expired
Components:	● SmartphoneApp ● ApplicationServer: ○ Dispatcher ○ RechargeManager ○ NotificationManager ○ Model

Requirements:	[R23] The system allows users to pay for a charging
Components:	● SmartphoneApp ● PaymentAuthority(external component) ● ApplicationServer: ○ Dispatcher ○ PaymentManager ○ Model

Requirements:	[R20] The system allows users to download a PDF of the ticket
Components:	● SmartphoneApp

Requirements:	[R24] The system notifies users about special offers [R25] The system notifies users about prices changing on the stations in which they have a reservation [R27] The system suggests users to go and charge the vehicle
Components:	● SmartphoneApp ● ApplicationServer: ○ Dispatcher ○ SuggestionManager ○ NotificationManager ○ Model

Requirements:	[R28] The system allow user to start a recharge [R29] The system allow users to attach the electric the vehicle
Components:	● SmartphoneApp ● ApplicationServer: ○ Dispatcher ○ SocketSoftwareManager ○ RechargeManager ○ Model

5. Implementation, Integration and Test Plan

5.1 Overview

The last chapter describes the implementation of the system, how its components have to be integrated and the methods to validate and verify. It is important to note that, as the computer scientist Edsger W. Dijkstra said: “program testing can be used to show the presence of bugs, but never to show their absence”. Therefore, the aim of the tests implemented for the system is to discover the majority of the application’s bugs before every release. Integration and implementation are strictly correlated, hence it often happens that the integration order coincides with the implementation one, that’s why although this first chapter (chapter 5.2) is dedicated to the implementation strategy, it happens to keep into account the integration test plan when defining it. Finally, it is very important that the code written is well-commented and documented (eventually with the help of some tool, such as Javadoc).

5.2 Implementation plan

The system will be implemented following a combination of the bottom-up and thread strategies, so as to combine the pros of both approaches. On one hand, using a thread approach allows to produce intermediate deliverables evaluable by the stakeholders; this is very helpful when it comes to the validation of the realized system. Whereas, on the other hand, exploiting a bottom-up strategy promotes an incremental integration, which facilitates bug tracking thanks to the possibility of testing the intermediate results (the sub-systems resulting from the integration of components) bit by bit as additional modules get integrated.

The thread strategy implies the identification of the features offered by the system and the portions of the components (that for practicality will be also referred as sub-components, even though they do not necessarily match with design sub-components) responsible for delivering such features. For a single function different subcomponents cooperate in its realization, therefore it is necessary to define an order for their implementation. To tackle this issue the above mentioned bottom-up approach is used.

This implementation strategy allows the task of implementing different features to be assigned to independent development teams that work in parallel. However, before doing so, it is important to spot eventual common components, this way to avoid producing the same component or subcomponent twice

5.2.1 Features identification

The features to implement in the system are extracted directly from the system requirements. It is important to notice that some of them require the implementation of entire components, while others require the implementation of more or less small portions of them. In the following there is a brief recap of the features of the system.

[F1] Sign in and sign up
These features are two basic functionality that can be implemented in a common way. It is important to make distinction between customers and operators but the interface and the interaction is the same for both parts. The only difference is the table in which to save the result and to make a query.

[F2] Creation of reservations
This is one of the core features of the system. It can be divided into multiple sub-features that can be developed separately: selection of station and book of a recharge. The first one is required for the implementation of the other because for the creation of the reservation it is necessary to already have selected a station. Furthermore, the selection includes the maps navigation and the search station features.

[F3] Recharge management
This is another core feature of the system. It contains the set of functionalities necessary to manage the effective recharging process, including the computation of the validity of the QR-code and the payment, but not the user’s notification management.

[F4] Station manager features
The station manager's features is a set of features related to the station's management view, including information regarding the stations and the features that allow a CPO to handle all the operations that regard the owned stations and to add a station to those owned. To implement all the functionalities for an operator it is mandatory to handle also the DSO features and the energy acquisition management.

[F5] User notifications
This is the feature that manages the notifications on user’s smartphones.

[F6] Smart suggestions
This is the feature that manages the suggestion on user smartphones about smart reservation.

Notice that the majority of those features require the communication between application server and client, therefore a portion of the Dispatcher has to be developed for each one of them.

5.3 Component Integration and Testing

In the following section, we specify for each stage, what components are implemented in the stage, and how the components are integrated and tested (there is an integration diagram for each stage).

In the first step [Figure 5.1] the Model is implemented and unit tested using a driver for the components that are still under implementation. These components are responsible for all the interactions with the database and are needed by the majority of all the other components.

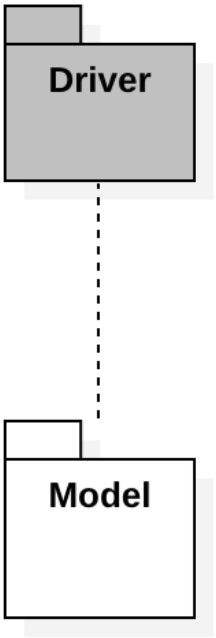


Figure 5.1 - Model developing

[F1] Sign in and sign up developing

Then we give priority to the development of authentication mechanism because some components need features developed in this stage. The EmailProvider component is not tested because it is an external component and so it is considered reliable. [Figure 5.2]

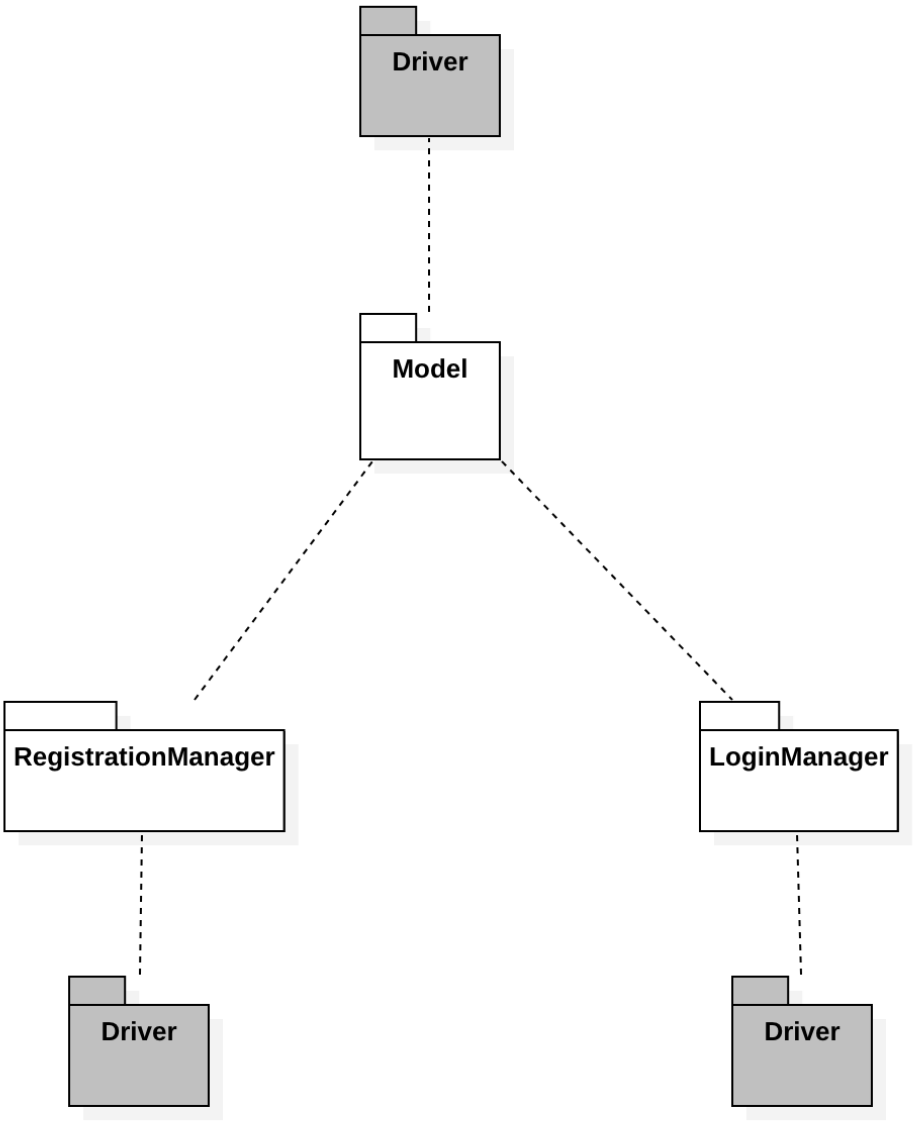


Figure 5.2 - Sign in and Sign up developing

[F4] Station manager features

We prefer to start developing these components instead of [F2] because some information is related to the reservation process as, for example, the functionality that checks whether a socket is available or not. [Figure 5.3]

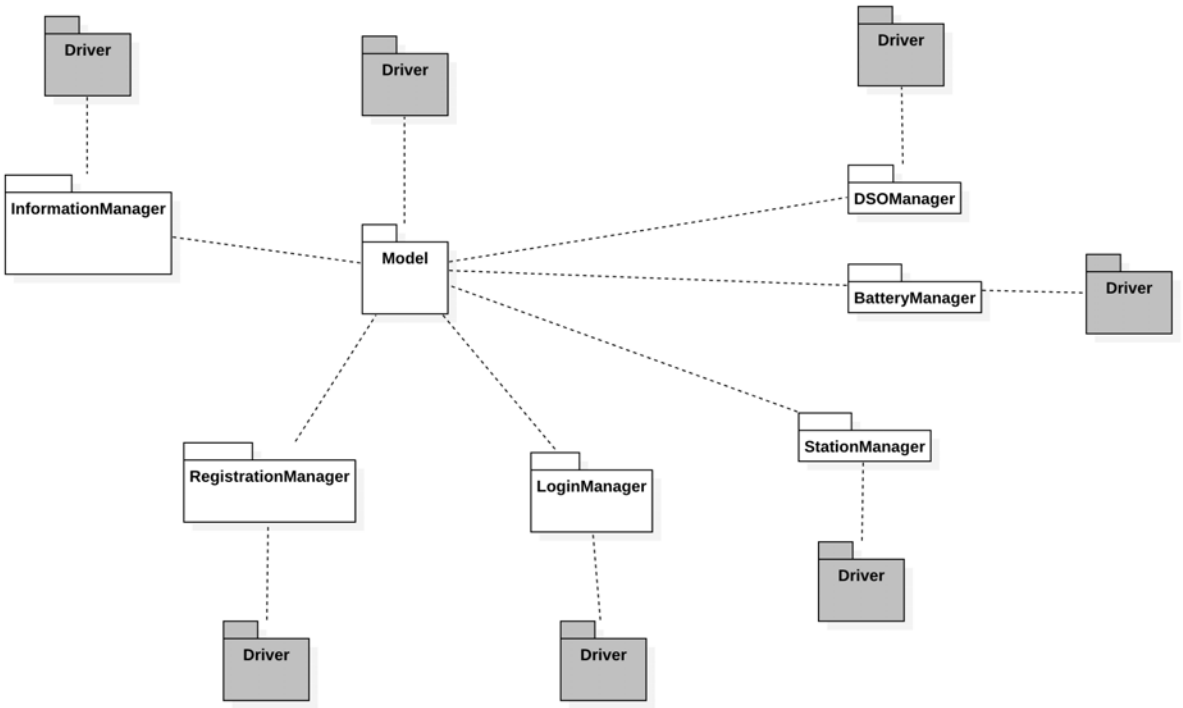


Figure 5.3 - Station Manager developing

[F2] Creation of reservations

Now we are ready to implement the testing part related to the booking of a reservation. This is one of the main features of our system so it needs to be well tested. [Figure 5.4]

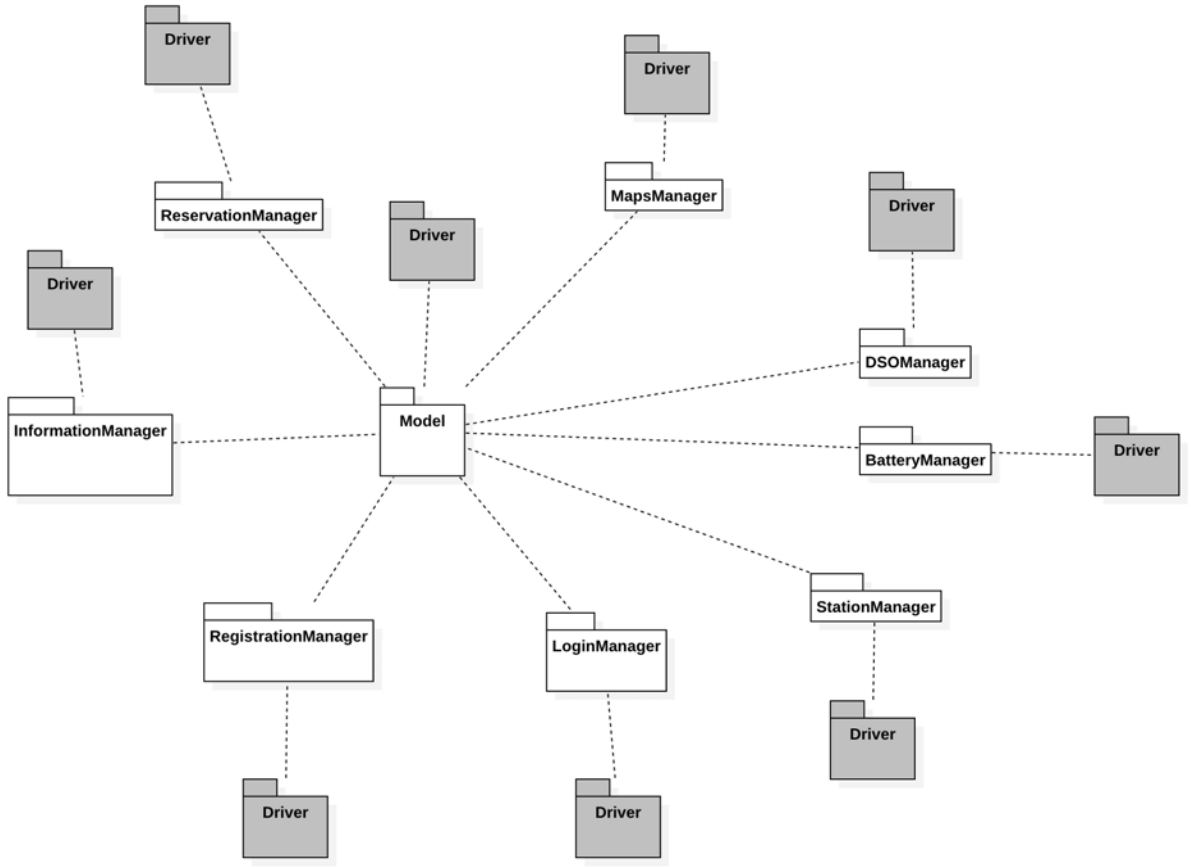


Figure 5.4 - Booking Reservation developing

[F3] Recharge Management

We add all the components that interact for the recharging process. So we will test not only the RechargeManager but also the components related to DSO features, in particular DSOManager and BatteryManager. [Figure 5.5]

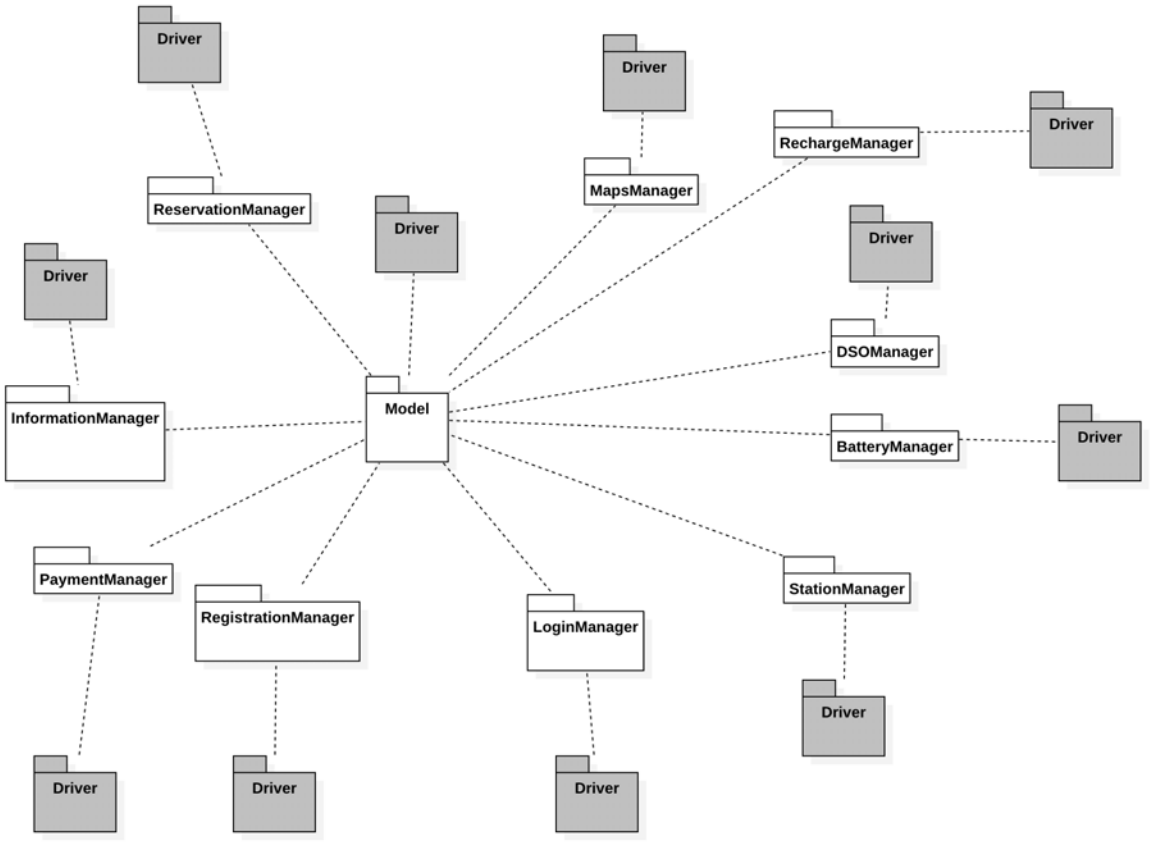


Figure 5.5 - Recharge Management developing

[F6] Smart suggestions

We have to implement the Smart suggestions component and part of the User notification component together because the first component uses the second to inform users about possible reservations. [Figure 5.6]

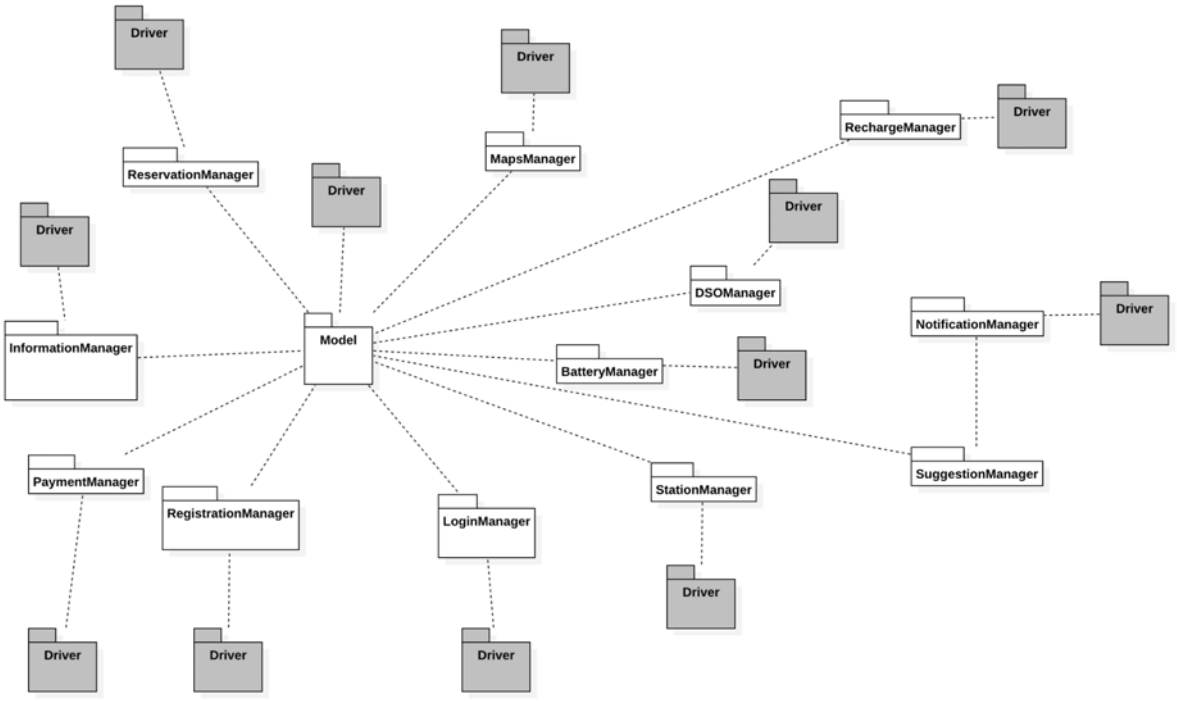


Figure 5.6 - Smart Suggestions developing

[F5] User Notification

Now the User Notification component is integrated with all its functionalities.To do this the component has to interact with RechargeManager and DSOManager. [Figure 5.5]

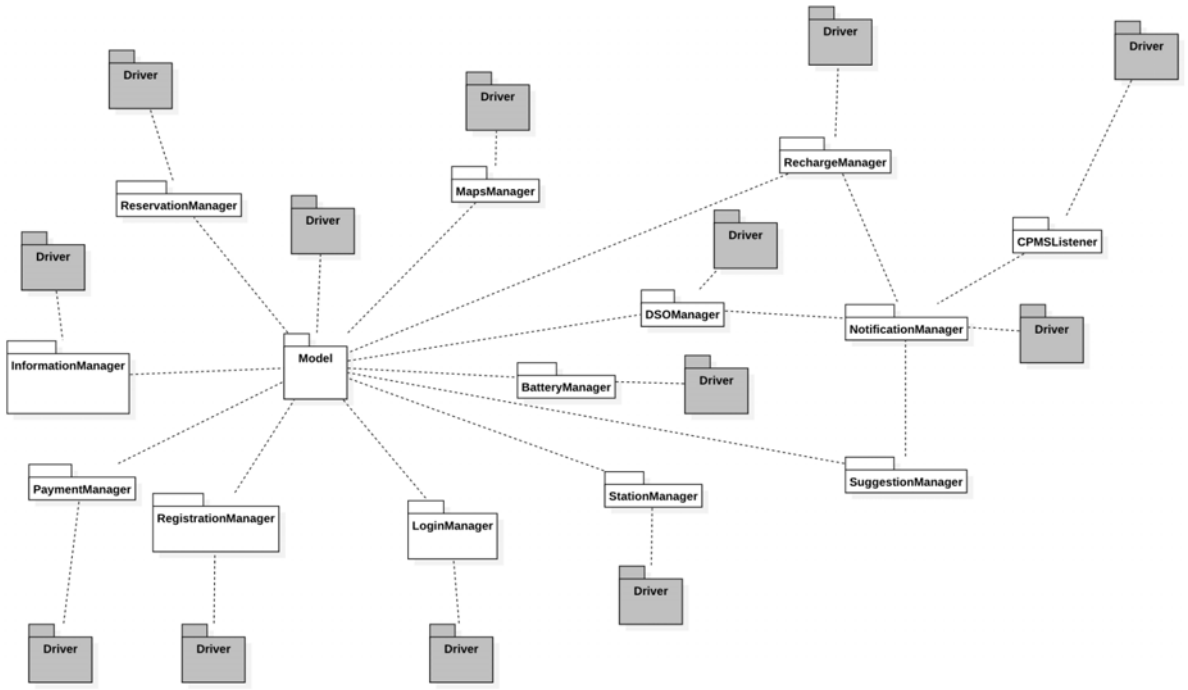


Figure 5.7 - User Notification developing

5.4 System testing

The eMall system undergoes a series of testing activities, each one of them with a different level of granularity and a different scope. During the development phase, every single component or module needs to be tested on its own to check whether it is performing as expected. To do so, since some components may not work in isolation, it will be needed to think and realize Driver and Stub components that simulate the behavior of the surrounding modules (both the expected and also unexpected conducts so as to check the component's robustness).
Once the testing of the single components has been carried out up to a satisfactory extent, it's time to integrate and test the result of such integration. In our case it uses a combination of the bottom-up and thread strategies for integration and testing, as mentioned in the previous chapter.
Once the system is implemented, unit tested, integrated and integration tested, it must be tested as a whole, to verify that all features have been developed correctly and that they comply with the functional and nonfunctional requirements defined in the RASD. In order to test the latter thing, it is necessary to do system testing. At this stage, the software should be as close to the final product as possible. This testing phase should not be performed exclusively by developers, being a black-box technique, it can therefore also include stakeholders. The following testing step is System testing.

- **Functional testing**: consists in verifying whether the system satisfies all the requirements specified in the relative Requirement Analysis and Specification Document (RASD). Furthermore, during this phase, it might be possible to think of new features that might improve the user experience.
- **Performance testing**: the purpose of this testing phase is to spot any bottleneck, inefficient algorithm(bottlenecks affecting response time, utilization, and throughput), etc... More in general it consists in finding inefficiencies that might affect the system's performance. It needs an expected workload and an acceptable performance target before testing.
- **Usability testing**: establish how well users can utilize the system (Web application and smartphone application) for carrying out tasks. Given the key importance for the system to be easy to use, usability testing needs to be executed.
- **Load testing**: To expose bugs such as memory leaks, mismanagement of memory, or buffer overflow. It also identifies the upper limits of the components. In order to test the latter thing the load until threshold must be increased and the system with the maximum load it can operate for a long period, must be load
- **Stress testing**: To make sure that the system recovers gracefully after failure. In order to test the latter thing, you have to try to break the system under test by overwhelming its resources or by taking resources away from it.

5.5 Additional specifications on testing

The system has to be tested whenever new functions are added to check the presence of bugs. On top of that, also the testing described in 5.4 has to be done, in order to verify the correct behavior of the system, during the implementation, in order to know as soon as possible if bugs are present.

During the development of the system it is also very important to receive feedback from users and stakeholders (in the first place who required the creation of the system, but also the end users). This should happen regularly, whenever a feature (or a part of it) has been implemented. First, a description of the functionalities should be given to them, so that they can check if the system is going to be what they expect. Then, some alpha versions of the system should be provided to them in order to receive feedback regarding eventual malfunctions and most importantly their level of satisfaction with
how the system is developed.
Doing so will contribute to validating the system during the creation process, and this is critical. This way developers know as soon as possible if the system they are creating is correct, and can eventually change some parts if its implementation does not reflect the cusotmers' expectations.

6. Time Spent

The time tables written below represent just an approximation of the time spent for the writing and discussions the team had for each specific chapter of this document. These times have not been measured while producing this document and are just based on the personal perception the team members have of the time spent.

[redacted]

Chapter	Effort(in hours)
1	1
2	52
3	1
4	2
5	4

[redacted]

Chapter	Effort(in hours)
1	0
2	2
3	30
4	1
5	20

[redacted]

Chapter	Effort(in hours)
1	3
2	50
3	0
4	4
5	5

7. References

- Mockups made with: moqups.com
- Testing models and component architecture made with: StarUML
- sequence diagram models made with: StarUML
- High level architectures are made using www.draw.io
- Software Engineering 2 course material - Politecnico di Milano 2022-2023